UNIVERSITY OF OSLO
Department of Informatics

Metadata and robots for web analysis

Master Thesis

Nils Mathisen

November 2013

Abstract

Robots for the Internet are computer programs that perform automated tasks online. Their functionality often involves interpretation of descriptive data, sometimes referred to as *metadata*, which can be embedded in web content.

The goal of this master project is to create a small set of robots for the Internet, and a command and control program that can manage their execution. Furthermore the software will have a graphical user interface, as well as functionality for storing and exporting the collected data.

To that end, the task starts with a study of the field of web analytics, to find out what software tools already exist, and then attempts to use this knowledge to design a new program. The software produced is not for general collection of online data, but for collection of data that can be useful in a research context, and for analyzing trends in how information is represented on the Internet.

The resulting software is extensible, and can be used to analyze a broad range of aspects of web pages. It includes scheduling functionality, allowing the user to configure when the various robots should be activated, so that the software can collect data independently, over longer periods of time.

Keywords: semantic, web, analytics, robot, bot, Internet, crawling, spider, metadata, meta, tag

Acknowledgments

First and foremost, I wish to thank my supervisor, Gisle Hannemyr, from the Institute of Informatics. He provided the ideas for a project that allowed me to combine several of my favorite topics. Throughout the project he has always found time to support me, and all my questions, whether complex or naive, have been met with reflected and thorough answers.

My friends and family have also given me a lot of support, both in the form of encouragement, as well as advice and suggestions. Thank you, Aslak, Joakim, Janne, and many others. A special thanks goes to Jan Mathisen, for invaluable help with statistics, Excel, and visualizations.

All mistakes made in this project are of course my own.

Contents

List of Figures

List of Tables

Glossary

Academic Web Watch The name of the software developed in this master project.. xi

ACAP Automated Content Access Protocol. xi, 21

AWW Academic Web Watch. xi, 38–43, 45, 50, 58, 68, 74, 88, 91–94

Bot Common short form for web robot.. xi

CCC command and control center. xi, 3, 32, 37, 39, 40, 42, 43, 49, 50, 54, 58, 59, 62, 78, 96, 100

ccREL Creative Commons Rights Expression Language. xi

CDATA character data. xi, 17

Command and Control Center The part of a software program that controls the flow of execution of smaller modules, in coordination with users or scheduled tasks. xi

DoS attack denial-of-service attack. xi, 25

GUI Graphical User Interface. xi, 39, 77, 94, 100

HTML Hypertext Markup Language. xi, 11, 15, 16, 19

MVC Model View Controller. xi, 43

ORM Object-relational mapping. xi, 48

RDFa Resource Description Framework-in-attributes. xi, 1

Robot for the Internet Throughout the text there are many references to *web robots* and to *robots for the Internet*. The two terms are often references to the same type of computer program. Robots for the Internet is a more general term, but web robots is a more commonly used term. While robots for the Internet do not always operate on the web, robots for the web are a type of robots for the Internet.. xi

Web robot (See *robot for the Internet*.) A robot for the Internet. The use of the word *web* could indicate that the robot operates on the World Wide Web, but it is a loosely used term. xi

XHTML Extensible Hypertext Markup Language. xi, 16

Chapter 1

Introduction

1.1 Motivation

The World Wide Web has been available to the general population for a couple of decades, and since its early stages it has been searchable, by the use of various search engines, and software tools. However, it was initially problematic to search it in intelligent ways. The software performing the searching had few means to judge the relevance of search results, other than basic techniques for text comparison. This may be one of the reasons why, around 2001, the vision of the *semantic web*[4] emerged.

The semantic web is a concept involving advanced descriptions of web content. These descriptions can be interpreted by software programs, and that made it possible to search and analyze the Internet in other ways. It meant that programs could make more independent decisions in the categorization of web content, and participate more actively in the sorting and navigating of that content.

Here comes a master project about analyzing the Internet. In spite of all the tools in existence, there are still gaps in the available software functionality. In web analysis for research, one may be interested in a particular set of details, from selected online locations, at selected points in time. For example, an experiment may require to monitor the changes in the amounts of Resource Description Framework-in-attributes (RDFa) code used in ten popular newspaper-websites, over a period of six months. There are tools available for this type of search, but they are less developed than tools for a general web search.

While there has been a significant amount of research into technology for a general web search, it would also be useful to have more research into technology for extracting precise and dynamic information from the Internet,

in highly customizable ways. New software tools could be made, that enable users to specify what data to download, and when to do so, with more options than general search tools provide.

1.2 Research focus

In short, the task is to study the field of web analytics, and summarize the types of analysis tools that are available as free software. When that is done, the information will be used to design and implement a set of web robots, and a software framework to handle their execution.

The resulting software should cover a broad range of aspects of web pages, and execute at set intervals, over longer periods of time.

In other words, this project will focus on the use of metadata and web robots in the context of web analysis. The details around this are described in the chapter on research methods (chapter 3), but below are the main research objectives.

R1 Study the field of web analytics and web robots, and summarize what exists.

R2 Conduct experimental design by defining and creating a Command and Control Center (CCC), a Graphical User Interface (GUI), and a couple of sample robots.

R3 Conduct experiments with the sample robots, and produce data and visualizations.

R4 Analyze the findings from the previous activity, and discuss the usefulness of robots in web analytics.

1.3 Document outline

This document comprises seven chapters, as described below.

Introduction In the first chapter there is a brief introduction to the field of web analytics, and to specific topics for which further research could be valuable. This forms a motivation, which in turn is used as a guide to formulate a research aim for this project, as well as individual research objectives. At the end of the chapter there is an outline of the whole document, with a short description of each chapter.

Background is the name of the theory chapter. It has three main sections, giving an introduction to web analytics, to the semantic web, and to web robots.

In relation to the semantic web, several common standards for metadata are discussed, with a focus on their intended use, their syntax, and how they are integrated into web content. Then a typical architecture of for web crawlers is described. This is followed by a discussion of common features of web robots, such as *politeness*. The text continues with a discussion of several implementations of web robots. They have user groups of various sizes, and most of them are available free of charge.

Research methods This is where the various project-related tasks are described in detail. The research objectives are discussed more thoroughly. Then, the section on *research strategy* follows, where useful research methods are selected. The intended use of these methods is then described for each of the project tasks.

Data collection will be performed in the form of program execution, with multiple configurations. Some details of this must be left for the design and implementation chapters, but a general description is given at this point.

A plan for evaluation of the collected data follows, and lastly a few comments are given about limitations and potential problems.

Program design A design proposal is created, resulting in a program that should implement all the functionality specified in the introductory chapters. The chapter starts with an overview of prerequisites, and available resources. Then programming languages are chosen. When these choices are made, more fine grained details can be resolved, such as which programming libraries to make use of, and what types of interfaces, control structures, etc. can be used in the design.

The robots are created as extensions for the command and control center (CCC). While these extensions are an important part of the project, a good deal of the programming effort will concern the CCC, which handles their execution. Therefore, a large part of the design chapter also revolves around this topic. Details are specified for how extensions are added, how users can control them, and how their execution can be scheduled. Solutions are also found for the storage of datasets and of their descriptions.

When this has been taken care of, work is started on designing the two main robots that are included in the software. Part of this work consists of describing the functions all robots must implement, in order to interact with the rest of the software, but there are also detailed explanations of the

individual features of the robots.

The main functionality of the program will initially be controlled by a command line interface. When all other components have been designed, an attempt is made at designing a graphical user interface as well.

At the end of the chapter there is a section about portability, as portability is expected to be beneficial to an open source project. There is also a short discussion about *thread programming*. It was first attempted to avoid use of threads in this project, to keep the the control flow as simple as possible, but during the design process it becomes more relevant than first assumed.

Implementation Here there is information on inclusion of *third-party* robots in the project. The next topic is data collection, and the address collections used for this purpose. Some effort has been put into creating address collections representable for the Internet as a whole, but it was found to be more challenging than first assumed. An attempt to generate random IP-addresses automatically is described, followed by a discussion of the quality of the resulting address collection. As an alternate approach, a manually constructed URL-collection is also created. The latter should be sufficient for a reliable analysis, as long as care is taken when generalizing over it.

With the collection of web addresses in place, the next topic is the data collection itself, and details about its execution. This includes descriptions of both scheduled and manual execution of robots, depending on what types of data is collected. After the collection has been completed, the collected data is used as input to the various visualizations.

At this point in the text, most of the program features have been implemented, and the focus changes to evaluation of the program. Afterwards, there is a description of revisions made as a response to the evaluation. Then the chapter concludes with a note on program dependencies for the finished version of the software.

Findings In the findings chapter the output of the data collection is described. First the datasets and the data collection period are commented on. Then the datasets are used for generating tables and visualizations. These are then explained, and comments are made about what they can tell us of the web pages they describe. The chapter includes a discussion of the output from the data collection, and of the quality and usefulness of the software. The software is evaluated based on how it performed during the data collection.

The research goals are evaluated, and shortcomings are pointed out. It is described where the development has taken different directions from what

was expected, and the usefulness and usability of the finished product is reviewed. This is then summarized, with a conclusion of the project work as a whole.

Final word The final chapter is a comment on how the work has progressed, with some personal reflections about the experience. It contains information about what will become of the software later on, and discusses whether the quality of the software is high enough that other developers and users might take an interest it. This is followed by suggestions about what kind of improvements would be most useful.

1.4 Chapter summary

That concludes the introduction. A motivation has been given for the work that is to come, followed by a definition of research goals. Then there was an outline of the entire document.

Among the chapters to come there will first be a review of literature on metadata and robots for the Internet, then a method chapter which outlines how the required work will be done. The next chapters describe the program design, and the implementation process. When the implementation has been completed, the findings will be discussed.

Chapter 2

Background

This chapter contains a literary review, that introduces the background material relevant to the project. In accord with the first research goal, described in section 1.2, this chapter begins with a study of the field of web analytics. As this field is explored, we arrive at other topics that are relevant to the project, namely the semantic web, metadata, and robots for web analysis.

By the end of the chapter it is hoped that enough has been learned to understand our problem domain, and to make sound decisions during the design process that will follow.

2.1 Web analytics

Before describing the topic of web analytics, a few words will be said about the meaning of the term itself, to clarify the relation between the words analytics and analysis. For the word analytics the most relevant definition found was the following:

> *1. a. Chiefly Philos. The science or method of using analysis to examine something complex; spec. the branch of logic that deals with analysis, esp. with reference to the book of the same name by Aristotle.[...]*
> *b. The collation and analysis of data or statistics, esp. by computer, typically for financial or commercial purposes; the data that results from this; (also) software used for this purpose.[...]* [1]

[1] The Oxford English Dictionary: `http://www.oed.com/view/Entry/273413?redirectedFrom=analytics#eid` (visited on 20th of October, 2013)

The definition indicates that analytics can simply mean the use of analysis, possibly involves the use of computers, and is often used for financial purposes. In addition it carries connotations to Aristotle's works about logic. The difference in meaning from the word analysis does not appear of great importance to our context, but the word analytics is used because it is popular in relation to analysis of the web.

Among the articles available online, web analytics appears to mostly concern site owners' analysis of traffic on their own web sites. There is a natural reason for this. For most people, it is not trivial to acquire data about activity on other people's web sites. Secondly, in a business context, the popularity and performance of their own web sites is a main concern for the majority of commercial companies on the Internet.

In our context, web analytics will be viewed as a general term, including topics such as analysis of web content, usage patterns for web sites, the traffic flow of data packages, and other related online phenomena.

2.1.1 Methods used in web analytics

Web analytics is a large field. In this project will be limited to the types of analysis that involves the use of web robots. However, datacollection through the use of cookies, and proxy servers are also common in use.

Cookies Among the types of data collection mentioned in the previous paragraph, the use of cookies may be in most widespread use. It differs from the other approaches, in that additional information is exchanged between user computers and web servers.

> *Web-based applications often use cookies to maintain state in the otherwise stateless HTTP protocol. As part of its response, a server may send arbitrary information information, the "cookie," in a Set-Cookie response header. This arbitrary information could be anything: a user identifier, a database key, whatever the server needs so it can continue where it left off.* [13, p.153]

In other words, cookies can allow web servers to save information on the computers of the users that request web pages from it. The maintenance of a state, on the those computers, means that servers can add and retrieve information, with details about the user and his/hers browsing history.

This solution will often require less resources than the other approaches, because it is only necessary to configure a web server to send and retrieve cookies.

Cookies can be categorized as first-party, or third-party, as described in the quote below.

> *In the early use of cookies, users could assume that the site the user had connected to would be the site the cookie would be returned to. As cookie technology developed, third-party sites began to make use of cookies, typically through the use of banner ads but also through the use of one-pixel images, effectively invisible to the user. Thus, users could no longer assume that a cookie apparently being sent from the visited site would be returned to that site.* [16, p.49]

While first party cookies are exchanged between a user's computer, and web site that the user has navigated to, third party cookies may end up on other web sites than the one that has been visited. This can be a problem for privacy protection when the technique is used to collect personal information.

Google Analytics Google Analytics is a *cookie-based* tool for web analysis. Code inserted in the site owner's web pages causes traffic information to be passed on to Google Analytics for registering.

The software is aimed at web developers who need information about what consequences design changes have on popularity, and about which parts of the sites the public find their way to.

In relation to first-party and third-party cookies, Google appears to use mainly first-party cookies. However, in the case of what is referred to as *Google Analytics for Display Advertisers*, Google's web pages explain that *a third-party DoubleClick cookie is used*[2]

Robots As mentioned, web robots can also be used for analysis. While the use of cookies can be less demanding to implement, one valuable aspect of web robots is that they are not limited to analysis of sites you have access to, because, in most cases no modification of the site content is necessary. No code has to be added to web pages, for the robot to function. In some cases, a robot can simply be informed of which site to analyze when it is activated. Therefore, unless you have access to proxy servers for analyzing web traffic, robots are one of the most efficient means for analyzing 3rd party web sites.

In this project, the data collection will be done through the use of robots, and towards the end of this chapter their functionality will be given a more thorough description.

[2] `https://developers.google.com/analytics/devguides/collection/analyticsjs/cookie-usage`
(visited on the 26th of October, 2013)

Proxies Proxy servers can be used to collect information about packages sent across the Internet. This can be used to analyze traffic flow in the geographical region of those servers, but you can also collect information about the types of packages that are handled by the server. Analysis based on the use of proxies is more valuable if you have access to a network of servers, spread across a larger area. This is something average web site owners do not have.

In 2011 an article was published, named *Towards Understanding Modern Web Traffic.*[12] It presented results based on analysis of data from a globally-distributed proxy system. This system is referred to as the CoDeeN content distribution network (CDN).[12, p.296] The article states that it *serves over 30 million requests per day*[12, p.296] and is available to anyone[12, p.296]

The data set the article made use of was explained to include *browsing behavior of over 70.000 daily users from 187 countries*[12, p.295], and this made it possible to conduct a wide variety of analysis. For instance, it was confirmed that more complex web pages have resulted in significant downloads occurring after the initial download of web pages.

> *...half the traffic now occurs not as a result of initial page loads, but as a result of client-side interactions after the initial page load...* [12, p.296]

Proxy servers are best suited for analysis of certain topics. They can be used for analyzing traffic flow, or for comparing different types of traffic. Among other things, the latter can give indications about what users are interested in, as shown in the following quote:

> *... Brazil (not shown) shows a higher fraction of uncacheable XML and audio traffic than other countries. This is due to the popular use of real time update of sports games and live streaming of audio.* [12, p.304]

More general changes in the usage of the web were also found. For example there is increasing use of multimedia content, and dynamic content, as well as traffic related to web analytics.

> *... Flash video and Ajax traffic is consistently increasing, and search engine/analytics sites are tracking an increasingly large fraction of users.* [12, p.307]

2.1.2 Topics for analysis

For commercial organizations, who want to measure the quality of their web sites, a set of metrics must be chosen. These metrics should consist of things that are convenient to measure, and which are expected to give a picture of the web site activity that can be used for analysis from a business perspective. Which metrics that are seen as useful, is a large topic in itself.

> *Over time, businesses have begun to find the use of basic metrics such as hits and pages views to be woefully inadequate for assessing the success of Web sites, due to the fact that their simplistic and ambiguous nature can induce misleading conclusions (for example, the spidering of a Web site can indicate lots of hits, the use of frames can greatly increase the number of page views, etc.)*[18, p.285]

The quote shows that some of the factors that are easiest to measure may not be useful for analysis. The first example says that, since much of the online traffic is caused by web robots, the number of page downloads does not indicate the number of human users. The other example mentions the use of frames in Hypertext Markup Language (HTML) code, which is a technique that combines multiple web pages into what can look like one single page. Viewing this combined page means downloading multiple HTML files, and if parts of the combined page are updated it may be counted as a page download, even though only a small part of the display is altered.

The article goes on to promote the need for *advanced web metrics*, which they describe in the following way:

> *Advanced web analytics aims to measure and understand the relationship between the customer and the web site.*[18, p.286]

After this, the article introduces different approaches, where the type of organization, and its customers becomes an important part of the analysis.

The analysis of companies and their customers seems to have much influence on the development of web analytics. But in a scientific setting, web analytics may have an entirely different purpose. Consequently, web analytics for scientific purposes may be more likely to require custom made software tools.

2.1.3 The structure of web content

After examining the concept of web analytics, we are now familiar with common topics for analysis in the context of the web. In order to propose new

forms of web analysis, it is also important to be familiar with the different ways that web content can be structured. Standards for structuring and describing web content have changed considerably in recent years. Today web content is more dynamic than before, changing appearance based on various conditions, like a users location, or preferred language. It has also come to include more semantic descriptions. This means that web documents contain information that can aid computers in classifying the contents. This development is often referred to as the development of *the semantic web*.

2.2 The semantic web

If computers can interpret web content on a semantic level, it enables them to categorize and process that content in a more efficient way. This is a key concept in the context of the semantic web.

Discussions of the semantic web often begin with a reference to an article by Tim Berners-Lee et al., submitted to the Scientific American in 2001. In a way it represents a turning point in the development of the web. The article is titled *The Semantic Web*, and directly after the title you can find the phrase, *A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities.*[4, 35]

In the article it is discussed how a selection of internet standards provide a foundation for the development of something that is referred to as the semantic web. The word semantic is often understood as relating to the meaning of utterances. To some extent, this is also the case here. It signifies that web content can be augmented with descriptions of its type of content, in a way that can be interpreted by computers. Ideally this will enable computer programs to make judgments about the relevance of web content, depending on the context it will be used in.

For instance, if a web search is made for the word *java* and the results found contain information describing which language they are written in, the results returned by the search tool can be made to consist only of web pages written in the user's preferred language. If more descriptive information is available in the results, then they may be filtered depending on whether they concern java, as in coffee, geographical locations named Java, the Java programming language, and so on. The search results can then be adjusted to match what is expected to be interesting for the user.

Figure 2.1 on page 13 shows technologies used in the realization of the semantic web. As can be seen, it still depends on older technology, but with layers of new standards added, that when used in combination provides new types of functionality. Lower layers, like the RDF layer, and the XML layer,

provide the means to create something known as an ontology. The ontology can be used for logic inference. Among other things, these inferences can be used for establishing proofs and trustability of web content.

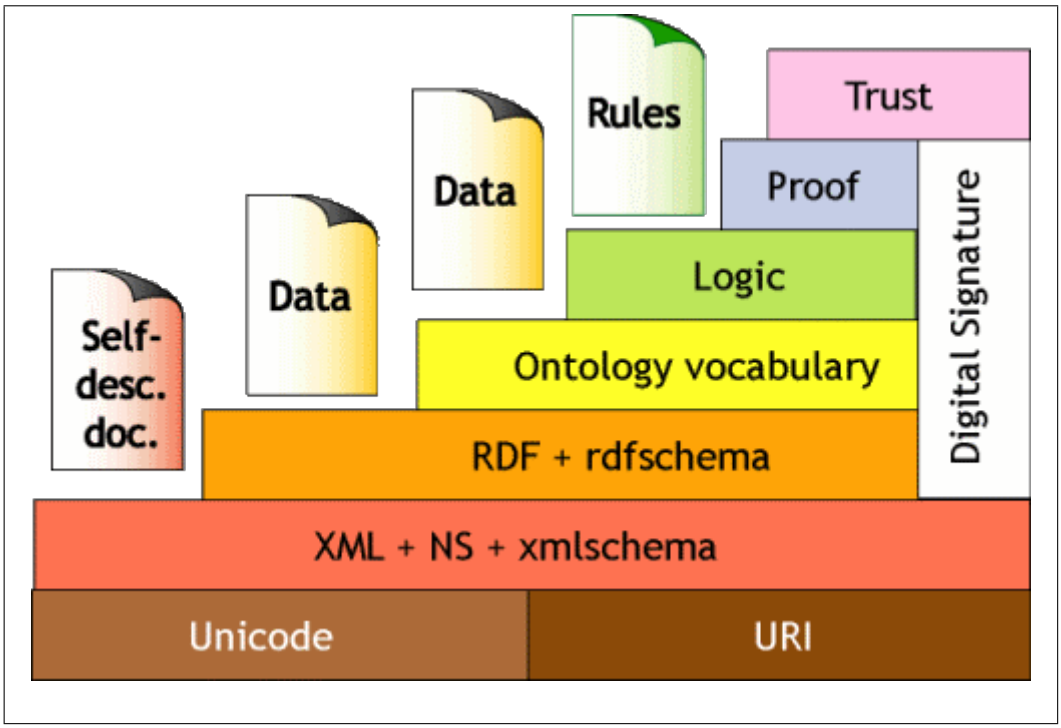

Figure 2.1: A representation of various layers in the architecture of the semantic web. Downloaded from `http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html` (on the 18th of October, 2013)
Tim Berners-Lee, Copyright © 2000 W3C, All Rights Reserved

Semantic web mining Traditional data mining is a way to *extract interesting patterns from homogeneous and less complex data.* [19, p.IIM] However, the semantic web requires that different techniques are used. The resulting is called semantic web mining.

> *Semantic Web Mining is a new and fast-developing research area combining Web Mining and Semantic Web.*[19, p.IIM]

One of the goals of semantic web mining is to make data mining more efficient by utilizing the features of the semantic web.

> *a) using semantic structures in the Web to enrich the results of Web Mining* [19, p.IIM]

Another motivation for semantic web mining, is that it may assist in the development of the semantic web itself.

> *b) to build the Semantic Web by employing the Web Mining techniques* [19, p.IIM]

Ontologies The term *ontology* is important to the understanding of the semantic web. In an ordinary context, the word can be defined as *The science or study of being*[3], but in discussions of the semantic web it is used differently. One article describes ontologies as containing the following.

> *...a set of knowledge terms, including the vocabulary, the semantic interconnections, and some simple rules of inference and logic for some particular topic.* [10, p.30]

An ontology provides tools that can enable software to make use of data in advanced ways. The ontology incldues a vocabulary, with names for the concepts that some system involves. There is information about how these concepts relate to each other, and there are logical rules, that software can use to derive new information from from existing information.

Standards used in the semantic web

On the Internet, the ontologies are realized through the use of metadata frameworks. In other words, various standards for marking up web content with metadata is used to describe how the contents relate to the various ontologies in use online. This is what makes metadata an important tool for web analytics, and for this master project. For that reason, the remaining part of this section will be used to introduce various standards of metadata.

2.2.1 Metadata

Metadata can take many forms, and there are many different definitions of the word. In this text it will be used as data that is used to describe web content. Sometimes metadata will be invisible to the reader, but embedded in a document, as a summary in the page header of a text, or in HTML-tags scattered throughout the text. Other times it will be readable both by machines and humans, as part of the actual text, but following strict rules

[3]The Oxford English Dictionary:
`http://www.oed.com/view/Entry/131551?redirectedFrom=ontology#eid` (visited on 20th of October, 2013)

for markup syntax, and thereby making it convenient for machines to process the document. Additionally, metadata can also be located outside of the file or content that it describes.

Several articles discuss the meaning of the term metadata. However, in relation to web robots the distinction between metadata and other data is not always important. Robots will often look at any type of document content, whether it is metadata or not. It is not interesting whether a part of a document is metadata or not, but rather if it can be used as a *selector*. By the term *selector* is here meant any data which can be used to locate sections of document content that a robot is meant to extract.

When collecting data using robots, we have other options than when doing it manually. It is faster, so we can afford to go through much larger quantities of text. We can work with any kind of file, as it does not need to be human-readable. The stricter the mark up of the documents, the better, but it is possible to work with plain text documents as well.

However, robots have the following disadvantage: You can not rely on common sense to decide whether something is relevant. How material is selected must be defined beforehand. This is challenging, because web content is often highly dynamic. If the web page in question is redesigned often, then structural changes are likely to create problems for navigating the content. One solution can be to search for for a printable version of the page, as redesign tends to have less impact on those versions:

> *If the information you want is available only embedded in an HTML page, try to find a Text Only or Print this Page variant* [9, p.5]

Another technique is to focus on as small a part of the web page as possible, making structural changes outside that region irrelevant.

There are numerous standards for metadata, and several with overlapping functionality. Which ones to utilize depends on the purpose of the software that is being built. Here follows a quick introduction to some of those standards.

RDFa

This section introduces the standard called *Resource Description Framework-in-attributes*, based on a paper named the *RDFa Primer* [3].

More and more traffic on the Internet is made from requests sent by web robots, instead of humans. Robots have different requirements to documents

than human readers. The structure of the document markup becomes more important. To meet this increasing need for *machine-friendly* documents W3C has worked out a framework for more detailed descriptions of document contents. It is described as:

> *...a set of XHTML attributes to augment visual data with machine-readable hints.* [3]

With these Extensible Hypertext Markup Language (XHTML) attributes, they hope to improve the machine-readable versions of documents. If software tools can have more detailed descriptions of web content, they will hopefully be able to parse and distribute it in more efficient and meaningful ways. Browsing web content is no longer so much about typing in specific addresses, as a process of letting automated programs pass you the information that they interpret as useful. Making use of software for locating information on the web seems to be a core feature of the semantic web vision. However, the RDFa is just a way of describing the content. This can be used for many purposes. Here follows an outline of its workings.

RDFa in brief The main idea of RDFa is to insert some attributes into already existing XHTML-tags, to augment human-readable data with machine-readable descriptions. This is done by adding *property-* and *rel-* attributes to XHTML-tags. A *property* is an URL. It can be an absolute path, or the document in question can have something like namespace defined. The property links the tag to a concept, in a taxonomy at that address. Then the rel-attribute is used to link HTML elements together.

This can be used to a greater or lesser extent, describing things of various complexity:

> *...markup for items as simple as the title of an article, or as complex as a user's complete social network.* [3]

At the time the article was written XHTML-documents with RDFa would validate, but HTML documents did not. This could be a problem for promoting widespread use.

After giving an outline of RDFa, the *RDFa Primer*[3] provides examples of usage.

The first example Inserting rel='license' into a link on a page, explains in machine-readable language that the link leads to a definition of the license for the current page. Thus a relation is described between two documents on the web, the current page and the page that defines the license.

The second example Insert property='dc:title' into a header tag. Here the argument to the property is a relative path, to a property described by the Dublin Core. Whether describing relative or absolute paths, all property names are URLs.

In a way this can help computers to *understand* what they are processing, almost on a semantic level. Semantics, or even understanding things on a cognitive level, is often described simply as knowing the relations between those things. Implementing this in web content would enable computers to perform complex tasks with the data. That is, if we can write software that handles this complexity in a useful way.

Flavor The *RDFa Primer*[3] refers to adding meta-info to web-pages as adding flavor.

Social network - the properties of a person There exists a RDFa vocabulary named FOAF, or *Friend-of-a-Friend*. In the article's example this is used in combination with the *typeof*-attribute to create a new data type, called Person. This type is then filled with attributes relating to people.

After this, the example in the *RDFa Primer*[3] explains how this vocabulary can be used to set up a structured version of an entire social network. It requires little code, and makes the RDFa appear as a highly efficient tool.

Types and nodes Two important concepts in the RDFa are *types* and *nodes.* A node can be created, and given a type. If a URL is not specified for the node, it is known as a blank node. Setting the node's type indicates what kind of data it will contain. There are predefined types, but it is also possible to create new ones.

Usability Creating complex metadata-structures through RDFa enhances the HTML with semantic machine-readable text. Having this kind of functionality embedded in readable character data (CDATA) could prove valuable. If human readable language and machine readable language are connected in that way, it may improve the usability of information on the Internet.

Still, the example-user in the *RDFa Primer*[3], appears to have more technical skills than should be expected from an average user. The standard is more likely to be successful if it is implemented on a software level, and thereby made more available, e.g. through a CMS for average users. This, however is not discussed in the *RDFa Primer.*[3] It describes how the metadata should be formatted, but not how it will be generated. It appears

important that insertion of the RDFa-metadata is automated. There may not be many people willing to write it out manually, only to adhere to a standard.

ccREL

Metadata can assist in many different aspects of handling web content. When it comes to administration and distribution of data, copyright is of essence. Here follows a discussion of *The Creative Commons Rights Expression Language* based on a paper by the same name.[2] The ccREL is a system for including machine readable copyright information in web pages. The objective is to:

> *...make it easy to publish and read rights expression data...* [2, p.32]

ccREL is Creative Commons' attempt at using RDFa for their purposes. Copyright is only a small part of what metadata is used for, but the principles are the same as in other contexts.

Creative Commons use W3C's technology. Both of these organizations are trend-setters, and not acting purely out of financial motivation. Copyright has long been difficult to express online. Material changes hands in different ways there than in the real world, and there are many small distributors.

CC provides easily embeddable copyright-notices, with simple categories of distribution. Previously, more research was required for including licenses in web publishing. With CC's solution, one only have to select one of their licenses and copy the appropriate code into the published documents.

By the use of ccREL it also becomes more convenient for software programs to process web content depending on copyright. The most obvious application for this is in web-search. If you needs to find free material, software can ensure that a search exclusively returns results fitting specified license requirements.

One place such a search can be made is at `http://search.creativecommons.org/` There, one can utilize a number of search-engines, automatically embedding license preferences along with search words in the requests.

A good design? ccREL is a fast and accessible way to incorporate copyright into web content productions. It is in widespread use. The paper about ccREL[2] gives us an example:

> *Flickr hosts approximately 50 million CC-licensed images (as of October 2007).* [2, p.21]

Microformats

Microformats are based on using old technology for new purposes:

> *Microformats are about using the standards we all know and love to convey as much semantic meaning as possible. [...] all you need to do to get started with them is familiarize yourself with the best ways to apply the tags and attributes you already use.* [7]

The quote, from the *Microformats Primer*[7], illustrates this by describing a business card according to the *vCard* standard, but doing it through the use of plain XHTML code. This is accomplished by using a collection of *div*-tags, that are assigned class names corresponding to the attribute names required by *the vCard* standard.

```
BEGIN:VCARD
VERSION:3.0
N:Çelik;Tantek
FN:Tantek Çelik
URL:http://tantek.com
ORG:Technorati
END:VCARD
```

Above is an example of a vCard, from the *Microformats Primer*.[7]

```
<div class="vcard">
 <a class="url fn" href="http://tantek.com/">
  Tantek Çelik
 </a>
 <div class="org">Technorati</div>
</div>
```

The HTML code above, also from the *Microformats Primer*[7], shows the same vCard, described through the use of XHTML.

Schema.org

`http://schema.org` provides collections of schemas that can be used in the markup of web pages. It is a collective effort, by companies behind several of the largest search engines.

> *A shared markup vocabulary makes it easier for webmasters to decide on a markup schema and get the maximum benefit for their efforts. So, in the spirit of sitemaps.org, search engines have come together to provide a shared collection of schemas that webmasters can use.* [4]

If the schemas are widely adopted, those search engines will be able to improve their service, and as large companies they have the opportunity to promote them. As is stated in the quote above, it would be beneficial to the search engine companies, because the more people adhere to the standard, the easier it becomes to create advanced algorithms for cataloging web content.

Schema is a young project, and a more clearly defined approach than older standards, such as Microformats. How it develops depends on how the web community responds to it. Few relevant academic articles were found, when searching for *schema.org*, so the research community does not yet seem to have become involved.

Dublin Core

The Dublin Core provides vocabularies intended to be used as metadata. Using these terms means classifying web content in categories common across the web. When many people use the same terms, it becomes more convenient for search engines to make use of that metadata. The name of the standard comes from a workshop in Dublin, Ohio.

> *The development of the Dublin Core metadata standard is guided by the DCMI. The DCMI had its beginnings in 1995 in Dublin, Ohio, where the first of the Dunlin Core Series workshops was held.* [8, p.41]

The standard is called *core* because the terms chosen are few, with broad meanings, which enable them to be used in a very wide range of contexts.

> *... Dublin Core metadata were intentionally designed to provide a basic set of elements that is both easy to apply, especially by non-library professionals, and suitable for a wide array of resource description communities.* [8, p.41]

Although an important focus appears to be on usage in web documents, Dublin Core can be used in many contexts, as it is readable by both machines and humans. The set of elements is small, but extendible.

[4] `http://schema.org/` (visited on 26th of October, 2013)

> *...Dublin Core was never intended to be a comprehensive metadata solution for all possible needs. Instead, it is more appropriate to view the 15 Dublin Core metadata element set as a common foundation upon which various communities can build more complex metadata schemes to meet their own specific needs.* [8, p.41]

If the elements of the Dublin Core is in widespread use on the Internet, then websites may be using some, or all of the core elements. In addition they can define their own metadata tags. If software is made in order to analyze the usage of Dublin Core metadata, it would be uncomplicated to compare the popularity of the core elements. The names of the elements that are added as extensions, on the other hand, would be more difficult to predict.

ACAP

The Automated Content Access Protocol (ACAP) concerns rights management for web content. It is a protocol that can be used by web publishers to mark up their web documents with information about distribution rights. This metadata can then be processed, for example by search engines. On the official web site, it is described in the following way.

> *It is a non-proprietary protocol, developed by publishers, which is designed to ensure that anyone who publishes content on the web and who wants to ensure that the web "crawlers" used by search engines and other online aggregators can read and understand the terms and conditions of access and re-use. In other words, ACAP is all about making copyright work on the web.*[5]

According to the above quote it appears that search engines and aggregators are expected to take an active part in copyright management, by filtering the web content they direct users to, based on how that content has been labeled.

The standard does not appear to be in widespread use. At the time of writing a search on `scholar.google.no` for *"Automated Content Access Protocol"* appeared to return very few, if any relevant results. This was also the case for a search on `emeraldinsight.com`. The scarcity of academic material about the standard indicates that its use could be highly limited. One way to investigate this further could be to crawl web sites, and look for presence of ACAP-code. This is a typical example of the type of tasks the software developed in this project could be made to perform.

[5] `http://www.the-acap.org/FAQs.php` (visited on 5th of October, 2013)

Robots.txt

The main purpose of the *Robots Exclusion Standard*, or *robots.txt*, is to inform web robots how to behave on websites. It should always be taken into consideration when *crawling* a site. On web forums, discussions can sometimes be seen about how polite the behavior of e.g. the *Googlebot* is. When a robot is created, it can be made to overlook the directions in robots.txt, but this can cause it to be banned from the site.

Sometimes a robots meta tag will be present in a HTML file, serving a similar purpose to that of robots.txt.

Metadata summary

We have reviewed different types of metadata for the Internet. Next we will examine the types and architectures of common web robots.

2.3 Web robots

We have now looked at several common standards for metadata. The purpose of this master project is to use that information to build robots for the Internet. Before this can be done, we must also look into what robots for the Internet are, and how they function.

Robots for the Internet are known by many names, and have various types of functionality. Robots for the Internet, robots for the web, or web robots, are fairly general terms, that often refer to software programs operating independently or automatically on the Internet. In this dissertation we will mostly use the term web robot, because it is short. The distinction between the terms the Internet and the web is not stressed.

Traditionally the word robot is often associated with mechanical machines that perform automated tasks, for example at an assembly line. This understanding of the word is strongly linked to a Czech play from 1920, by Karel Čapek. In English, the play is called *Rossum's Universal Robots*[5], and the Oxford English Dictionary explains the use of the Czech word *robota*, meaning *forced labour*, as leading to the following definition of the word robot:

> *Chiefly Science Fiction. An intelligent artificial being typically made of metal and resembling in some way a human or other animal.* [6]

[6]The Oxford English Dictionary: `http://www.oed.com/view/Entry/166641?rskey=CYWcKG&result=2&isAdvanced=false#eid` (visited on 20th of October, 2013)

If we do not look at their implementation, but rather at their functionality, the most important feature of robots could be said to be that of automation, or independent operation. When there is a task that is too big, or too tedious for people to perform, we can create something to do the task for us. If we include this in our understanding of the term *robot*, then the term also makes sense in a software context.

Web robots are not necessarily only information gatherers, but can also have sophisticated functionality for post processing of data. For this project, the robots created will both retrieve and process metadata and other information from the Internet.

2.3.1 Functionality and architecture

Some more examples of names referring to web robots are: *robots, bots, scripts, indexers, scrapers, crawlers, spiders, drones, scutters.* Several of these terms can be used interchangeably. Bot is short for robot. A script is usually a small sized program in a high level programming language. Neither of these have to be related to the Internet. An indexer is sometimes a program that goes through data and creates an overview of the contents. In the most extreme cases, like that of the *Googlebot*, this can mean sorting through a larger portion of the whole Internet, and may require days to complete, if it ever stops at all. A scraper may process acquired data, and extract something from it. Spiders and crawlers often perform similar tasks, and may use sophisticated algorithms for navigating the web (*crawling*), and searching out content.

Strict definitions of the different terms will not be made here, as they are often used freely and interchangeably, and because their functionality is often overlapping.

Besides describing their function, names of web robots often say something about the robot's purpose. Spambots and botnets are common names for malicious web robots. A spider may be a malicious robot, if it is used for scraping, and then republishing content. On the other side, if it is collecting information for a search engine, this is likely to be appreciated by site owners. Other categories of robots are political robots, votebots, financial robots, trading robots, search engine agents, knowbots, chatterbots, IRC robots, update detection robots, gaming robots, etc.

Web crawlers Web crawlers are a useful example for discussion. A possible architecture can be seen in figure 2.2 on page 24. Crawlers often work independently over longer periods of time. Sometimes they never finish, but

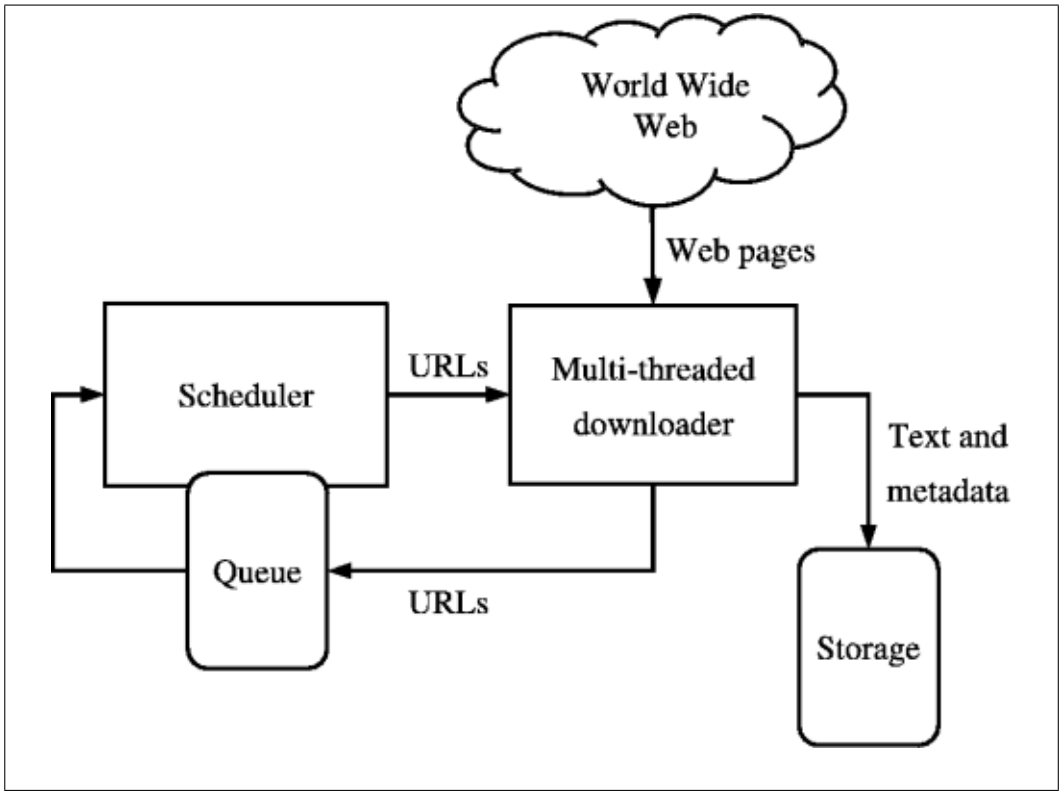

Figure 2.2: Possible implementation of a web crawler. The image is in the public domain, and was downloaded from `http://en.wikipedia.org/wiki/File:WebCrawlerArchitecture.png`

instead download content over and over, to check whether it has been updated. They are usually provided with one or more web addresses to use as a starting point. When they download what those addresses point to, more addresses are attempted extracted, and then placed in a queue. By having a scheduler that selects addresses from this queue, the crawler can keep downloading data, and exploring a partition of the web, until all suitable addresses have been used. Aside from extracting addresses from the downloaded data, it can be processed further in various ways, depending on the purpose of the crawler..

For different crawler different crawling techniques will be useful. For example an incremental web crawler can be used can suitable for indexing.

> *An incremental crawler [5], is one, which updates an existing set of downloaded pages instead of restarting the crawl from scratch each time. This involves some way of determining whether a page has changed since the last time it was crawled.* [14, p.403]

If the web sites crawled contain a high number of static (rarely updated) web pages, this approach can ensure that only the pages that have been updated are downloaded, and in this way reduce the network load.

Another optimization is the use of parallel crawlers. Parallel crawlers are crawlers utilizing multiple processes, in order to divide the work load. If you want to reduce the network load caused by crawling, the case of a distributed parallel crawler is relevant.

> *Distributed crawler: When C-proc's run at geographically distant locations connected by the Internet (or a wide area network), we call it a distributed crawler. For example, one C-proc may run in the US, crawling all US pages, and another C-proc may run in France, crawling all European pages* [6, p.126]

The word C-proc in the above quote refers to separate crawling processes cooperating to conduct a web crawl. Since the processes are geographically distant from each other, the network load is lower in the respective locations. How efficient this is, is described in the article as dependant on how the processes coordinate their work.

> *When C-proc's run at distant locations and communicate through the Internet, it becomes important how often and how much C-proc's need to communicate.* [6, p.126]

In other words, if the processes must communicate in order to decide whether to download web pages, this may create a significant amount of additional web traffic.

2.3.2 Politeness

When building web robots, and especially when building web crawlers, efficiency is an important criteria. Although it is important to be efficient, this efficiency should be tempered with politeness. By politeness is here meant that the robots execute in a way that is not provoking to owners of web servers, web sites, or web content.

An example of impolite behavior by robots could be to download a high number of web pages from one single server at a high rate. From a programmer's point of view this may seem like a minor detail, and one may hope that it will be overlooked. However, if it is done on a large enough scale it may create an unacceptable load on the server in question. In a worst case scenario, this could be equal to a denial-of-service attack (DoS attack), and

punishable by law. On a smaller scale, it may simply lead to the IP address of the crawler being blacklisted on the server.

If the behavior of common web robots is an indication, politeness is seen as something worth investing in. Below is a quote that supports this. It concerns the design of a well known web crawler, named Mercator.

> *Despite the need for speed, anyone running a web crawler that overloads web servers soon learns that such behavior is considered unacceptable. At the very least, a web crawler should not attempt to download multiple pages from the same web server simultaneously; better, it should impose a limit on the portion of a web server's resources it consumes. Mercator can be configured to obey either of these politeness policies.* [1, p.26]

It is common for web robots to supply a user agent string included with their HTTP requests. This contains a name describing the software making the request, possibly along with a version number. Not including this can also be seen as impolite behavior.

The robots.txt file described in section 2.2.1 can contain different rules for individual robots. To have an effect the directives contained in the robots.txt file depend on the robots being designed to obey them.[7] On the other hand, not adhering to robots.txt is also considered impolite, and may lead to the robot, or its IP-address being blacklisted.

Although requests from robots can be a burden for site owners, it also valuable to them, if it helps their site become more visible, for example in search engines.

Deep web No one have access to the entire Internet. When search engines are indexing the web they often try to comply with what site owners want to be visible in a web search. This means that a lot of the web is not accessible through average search engines. When referring to web content that is not discoverable through search engines, and public web sites, the word *deep web* is sometimes used. This type of web content may or may not be protected by passwords, or other security measures. What is important, is that it is not intended to be searchable. For example, a web site may generate its pages dynamically, from a database, but search engines will normally not try to explore the content of the database exhaustively. They will only go through the versions of these dynamic web pages that are pointed to by URLs that they discover while crawling the Internet. Attempting to explore web sites

[7] `http://www.robotstxt.org/faq/blockjustbad.html`
(visited on 9th of October, 2013)

on deeper levels, extracting content which there are no links pointing to, is likely to be disapproved of by the site owners. In certain contexts, it may also be illegal.

To sum up, here are some guidelines that should be followed in the creation of web robots for this project.

- Respect the instructions in the robots.txt file, and in W3C's Robot META Tags.[8]
- Limit the download rate of pages from individual web sites. When crawling multiple sites, it would be beneficial alternate between them, to reduce the load on the individual sites.
- The load on the local network should also be considered.
- Do not generate new URLs based upon pattern recognition extracted from analyzing existing URLs, or from URL query options.

2.3.3 Well known robots

Googlebot

Googlebot is the name of one of Google's crawlers, and it collects data that is used for indexing the web. Even if site owners have not registered their sites with Google, Googlebot is likely to find the sites, by following URLs linking to it on other web pages that it crawls.

Heritrix

Heritrix is a crawler belonging to the Internet Archive[9]. Development of the crawler was started in 2003, and it is presented as *an open source archival quality web crawler.*[17]

Mercator

Mercator is a *scalable, extensible web crawler written entirely in Java.*[11]

It is an old crawler. The quote above is from an article written in 1999, and few relevant results were found online, so the crawler may be in little use. However, the name is sometimes mentioned in relation to crawling algorithms (for example in section 2.3.2, about politeness). It was also included search products sold by AltaVista [11, p.14]

[8] `http://www.w3.org/TR/html4/appendix/notes.html`
[9] `http://archive.org/`

Webcheck

Webcheck is an open source tool that creates reports about web sites. It is written in Python, and presents its output as HTML pages. The web site for the project states that it is possible to add extra functionality[10]

Scrapy

Scrapy is a web crawler developed in the Python programming language.

> *Scrapy is an application framework for crawling web sites and extracting structured data which can be used for a wide range of useful applications, like data mining, information processing or historical archival.* [21, p.153]

As can be seen from the quote, the crawler has a many possible areas of application. The website for the project also informs that it is simple, extensible, and a has a healthy community[11]

Nikita the spider

According to the official website, Nikita the spider was discontinued in 2008.[12] It deserves mentioning because the website provides much information about how a crawler can be implemented in Python. The project is open source, and still available. The project also provides an alternative to Python's parser module for robots.txt files.

Robot catalog

There is no obvious place to refer to in order to get an overview of freely available web robots. Robotstxt.org maintains a robot database.[13] Searching the Python Package Index[14] for *web*, *robot*, or *web robot*, gives several results containing the words spider, crawler, or similar. The website cpan.org [15] provides a large, searchable archive of modules for the Perl programming language, and has categories for web related modules.

In addition a web search for web robots may return discussions recommending popular robots, but these will often be commercial, and the majority appear to be intended for general purposes.

[10] `http://arthurdejong.org/webcheck/` (visited on 9th of October, 2013)
[11] `http://scrapy.org/(visitedon24thofOctober,2013)`
[12] `http://nikitathespider.com/` (visited on 24th of October, 2013)
[13] `http://www.robotstxt.org/db.html`
[14] `http://pypi.python.org/`
[15] `http://www.cpan.org/`

2.4 Chapter summary

After defining the problems to be solved in this master project, we have now been through enough literature to start out work. This knowledge will be put to use in the next chapter, where decisions are made about which research methods to use.

Chapter 3

Research methods

3.1 Introduction to research methods

As described in the introduction, the research objectives are the following:

R1 Study the field of web analytics and web robots, and summarize what exists.

R2 Conduct experimental design by defining and creating a Command and Control Center (CCC), a Graphical User Interface (GUI), and a couple of sample robots.

R3 Conduct experiments with the sample robots, and produce data and visualizations.

R4 Analyze the findings from the previous activity, and discuss the usefulness of robots in web analytics.

Below follows a more detailed description of the tasks related to each of the research objectives. After that potential problems are discussed, as well as the tools and software to be used.

3.2 Study the field (R1)

The background chapter includes a study of web analytics, to help in getting to know the field. The semantic web is introduced, along with various standards for metadata used on the Internet. A closer look is taken at analysis conducted by the use of web robots, and several well known web robots are introduced. The method used for this study is reading of relevant academic literature, but also seraching the web and Wikipedia. It is a young research field, and some of the topics are not yet covered by peer-reviewed literature.

3.3 Conduct experimental design (R2)

In the design chapter it is decided what kind of software shall be constructed. The method used is experimental design, and to that end a selection of robots will be proposed, that can be used for for experimentation. A CCC will also be designed, that can control the execution of the robots. The CCC will also provide the users' interface to the system. The value of this part of the software lies in automation of tasks, by allowing control of several robots from one access point, and setting them up to run at specified intervals. The GUI will be an alternate way to access the CCC.

The goal is to gather a selection of web robots, with different types of functionality, that can be used in combination, for a wide range of tasks.

Some examples of interesting aspects of web pages are: page size, page complexity, content, growth, problems, quality, degree of semantic web functionality.

3.4 Conduct experiments (R3)

Two main robots will be designed. The first will create a time series, performing a relatively simple task against a high number or URLs, over a longer period of time.

The second robot will be less oriented towards change over time, but give a more complete picture of the current situation of web sites. This requires more complex functionality. First the robot must crawl the site, mapping out which web pages are to be gone over. Then it will investigate the pages found, recording degree and type of use of a selected standard of metadata. This robot can be run sporadically. It will not be included in the scheduled data collection, but be activated manually instead.

The first robot will look at specific web pages, the second at web sites as a whole.

The findings from these experiments will be used for discussion. Visualizations of the collected data will be created using Microsoft Excel.

Data collection

The software will run independently over time, and collect information about web sites. It will also be operated manually, for some parts of the data collection.

Program execution

The software will be run with two separate configurations, utilizing different types of web robots. One configuration will run periodically for a longer time period, collecting fairly trivial data, which is likely to change over time. The second configuration uses a more complex robot, which crawls entire web sites, instead of looking at single web pages. It will be executed manually.

The details of these robots will be specified in the design chapter. Their functionality will be chosen based on what was learned from the literary review, as well as on input from the supervisor of this master project.

URL collections used

When executing the software a collection of URLs must be supplied, giving the robots something to operate against. Which addresses are chosen will have an impact on our results. The focus is not on any particular subset of the Internet, but rather on general trends. Therefore the selection of URLs should ideally be representative for the whole of the Internet.

Automatically generated IP addresses will be used, if such a collection can be created in an efficient way. If it is found to be too time consuming, manually collected addresses will be used instead. The details around this are described in the implementation chapter.

3.5 Analyze the findings (R4)

The output from the robot execution will be used for discussion, as well for creation of visualizations in Microsoft Excel. The coding experience will also be discussed. The design and experiments will be the basis for an evaluation of the usefulness of robots in web analytics. The experience with development and application of the software will serve as material for a discussion of challenges and possibilities in similar projects.

Evaluation of the software

There will not be time to make large revisions in the program, but evaluation will be done of the quality of the design and implementation. Afterwards, it will be proposed ways in which the program can be improved.

Promotion of the software will not be a focus. It will require much time to implement all of the required functionality, and there will be little time left to make the interface intuitive for common users.

3.6 Limitations and potential problems

The main limitations are restrictions on time and equipment. There will be no traditional fieldwork, where the software is deployed with users who perform user testing. But the software will be tested in the real world, where it is used for collection of data from web pages for evaluation.

It is likely that some unexpected challenges will occur. The choice of programming languages and libraries will have a large influence on the outcome of the project. If the work is done in programming languages that have not been used before, then tasks will take more time. Unfamiliar languages can also turn out to lack features supporting the solutions that have been planned implemented.

It may also be problematic to achieve a data collection that is representable for the Internet as a whole. This depends on requiring a URL collection to use as input to the web robots, that is representable for the Internet as a whole. It also depends on avoiding any unintended side effects occurring during the data collection itself. If the robots collect data in a non-uniform way, the collected data may be compromised.

Should this happen, the results will hopefully still be usable, but it will be important to be careful when generalizing over them.

3.7 Tools and software

The work is mainly performed on the Linux based computers of the University of Oslo. Eclipse is primarily for Java programming, but was also found to work well with Python, after installing the plug-in *Pydev*. That proved difficult to set up with the standard installation of Eclipse, so another version of Eclipse was downloaded.

Version control and backup of the dissertation has been done with SVN against the Institute of Informatics' servers. For the software produced, GIT was used on SourceForge.

Running the finished product requires little preparation, aside from obtaining a basic Python installation. Some additional Python modules are used, but the main functionality should always be available. Support for visualization is a desireable feature in the program, but for this project visualization is done by exporting datasets as text, and importing them in Excel.

Online representation

A project is created on sourceforge.com. Although focus will not be put on building a user community, Sourceforge provides useful tools for version control, ticket management, and software distribution.

3.8 Chapter summary

That concludes the method chapter. The research objectives have been described in detail. Details were also given about how the program development and data collection will be performed. When these tasks have been completed, the output will be used for analysis and evaluation. By creating the software, using it, and analyzing the outcome, we will have gained knowledge that is then used to evaluate the quality of the software and the usability of robots in relation to web analytics.

The next chapter is the design chapter, where decisions are made about how the software should be constructed.

Chapter 4

Program design

The previous chapter described in detail the work to be done in this master project. In the current chapter there will be a detailed description of the structure and functionality of the software that is to be built. In other words, the work now starts on a design proposal. A choice will be made for a suitable programming language. Then the structure of the Command and Control Center (CCC) will be planned in detail. Solutions will be found for how it can relay communication between users and robots. Functionality is needed for registering and scheduling tasks in the system. There is also need for a programming hook, that can give extensions (e.g. robots) access to some form of storage functionality.

Although the main focus of this master project is web robots, the bulk of the coding done will be an investment in the CCC that controls the execution of those robots. The robots themselves will be lightweight in comparison. They will mostly perform single, clearly defined tasks, and each of them will only contain a small number of functions.

One of the reasons the web robots are lightweight is that details around their execution are managed by the CCC. Another reason is that common functionality for robots will be made available in a shared module, so that it does not have to be recreated each time a new robot is made.

4.1 Prerequisites

As described in the method chapter, the software will be developed on the machines at the University of Oslo. Portable solutions are used where possible, but the software is developed for use on Linux systems. The user interface will require some prior knowledge of how the program is meant to be operated. The installation process should be straightforward. Ideally the

software can be run directly after downloading and unpacking it. However, since the software will have dependencies it may not be so simple in all cases. Still, the goal is that it should be possible to install the program, and its dependencies, in a user's home directory, without any administrator privileges. This means that most users with access to a Linux shell can make use of the software.

A name for the program The program is named the Academic Web Watch (AWW). It is called *academic* because it is intended to be used for research, and *web watch* is used because it is a tool for monitoring changes on the Internet.

4.2 Programming languages and libraries

The Python programming language appears to be suitable for the system that will be developed. As it should be possible to integrate already existing robots, it is useful that Python has bindings for other programming languages, as well as functionality for communicating with other programs, for example via pipelines. Some robots are also available, that are written in Python.[1] Dynamic language features, such as passing function names as variables, are also useful when integrating third party code.

In addition, the language should have well developed functionality for web-related tasks. Three programming languages that are often mentioned in relation to web programming are Python, PHP, and Perl. Among these, Python stands out because it also provides well developed libraries for visualization and GUI-implementation. That could mean the entire main structure of the software can be implemented using one single programming language.

One case where Python is possibly not the best choice, is if the program should be web-centric. By web-centric is here meant that the system can be used via a web browser, from a remote location. In that case PHP could be a better choice. Web-centering makes it easy to give access to many users. It also means displaying results can be done almost anywhere. The drawback is a more advanced program installation. PHP requires a web server, and storing results requires access to an online database. Adding new robots would require support for uploading files. Still, a web centered solution is very appealing, but there are two main reasons why it will not be used. Firstly, the program should be easy to download and run, with as little overhead as possible. Secondly, focus will be put on supporting existing

[1] For example Scrapy: `http://scrapy.org/`

robots that are written in Python. The last could be combined with a web-centric solution, but is simpler with a standalone Python based program.

Regarding persistence, Object-relational databases appear useful. It would be convenient to work with an object model, but still have access to all the benefits of having a database underneath. However, a simpler solution will be used for this project. It is described later in this chapter, in the section named persistence (section 4.3.9).

Perl is also much used for web programming. It is a powerful tool for text processing, but using it for AWW could mean that some features must be implemented in other programming languages. It is possible to create Graphical User Interface (GUI), visualizations, and database functionality via Perl, but its strength appears to be text processing.

Graphics libraries

The GUI for the CCC will be discussed in section 4.6, but the possibilities of visualizing collected data should also be mentioned. Such functionality will not be implemented in this project, but the programming hook will be made in a way that it can be utilized both by robots as well as for the purpose of visualization. If extensions for visualization are developed at a later stage, Python has well developed libraries for this too.

The ImageMagick software suite has a Python interface, called PythonMagick. GNU-plot is available through Python. However, the Python package called EasyViz could prove even more useful:

> *Easyviz is a unified interface to various packages for scientific visualization and plotting. [...] Both curve plots and more advanced 2D/3D visualization of scalar and vector fields are supported. The Easyviz interface was designed with three ideas in mind: 1) a simple, Matlab-like syntax; 2) a unified interface to lots of visualization engines (called backends later): Gnuplot, Matplotlib, Grace, Veusz, Pmw.Blt.Graph, PyX, Matlab, VTK, VisIt, OpenDX; and 3) a minimalistic interface which offers only basic control of plots: curves, linestyles, legends, title, axis extent and names. More fine-tuning of plots can be done by invoking backend-specific commands.*[15]

As can be seen from the quote, Easyviz can utilize various backends, such as Gnuplot, Matlab, etc. This means that using Easyviz for visualization can be done without considering which visualization engine will be used. This gives increased portability.

4.3 Command and control center

This section will be used for a description of the CCC. The topics include program structure, program behavior, user interfaces, user commands, persistence, program tasks, scheduling, etc. But first some important concepts in the program will be explained.

Note on terminology: In the following discussion of the software a few reoccurring words need to be explained:

- Command: A text string, usually from the command line interface, or an instance of the class *Command*, containing information about a specific function call.
- Task: A command string, and details concerning when and how it should be executed.
- Robot: A python module/file representing a web robot, and conting required functions for being activated by the CCC. A robot is a type of extension for AWW.
- Visualization: A python module/file that can generate visualizations, based on collected datasets. A visualization is another type of extension for AWW. No visualizations will be made in this project, except for a template, but the program is designed to support this.

4.3.1 Program functionality

AWW will be created as a Python package. Starting the program in its default mode, will open up a user interface to the CCC.

The CCC will provide several types of functionality. It can be used to activate the various web robots included. It also can also be used to export or truncate the data sets that those robots produce. If visualizations are created for AWW at a later time, functionality will be in place in the user interface for activating them.

A simple scheduling functionality will also be included. The scheduler can be started from the user interface, and will run in a loop, which iterates a list of tasks, and runs their associated commands when the time is right.

The user interface supports the creation of tasks. A task can contain any kind of command that is understandable by AWW's command line interface. This means tasks can be used for robot activation, but also for other things, such as setting up automatic export of data sets. Lists of URLs can be read

from file, and stored in a task. When a task is used to activate a robot, this list can be passed to the robot as an argument.

It will be possible to extend the system, by adding extensions in the form of robots and visualizations. These extensions can be added to subpackages of the Python package AWW. In Python packages there should always be a file named __init__.py. If extensions are listed in the init file of the subpackages they are placed in, then AWW will import them the next time it is started, and call one of their functions, to make them register their information and datasets in AWW's database.

4.3.2 Standards and politeness

It will be attempted to make AWW's robots adhere to common standards and conventions for web robots, such as metatags, politeness, and robots.txt. Number of requests per minute, and bytes downloaded per minute will also be controlled

4.3.3 Modes of operation

By passing command line parameters at start-up, it should be possible to start AWW in different modes. There will be a command line mode, a GUI mode, and a non-interactive mode which only starts the scheduler. In addition there will be a mode that executes one single command, given along with the parameters, before exiting.

4.3.4 Communication between users and robots

The robots will be relatively independent. What this means is that they are commanded to start, and then they operate on their own. A robot's creator can program it to do most anything possible within the Python language, and include additional modules as needed. Robots made in other programming languages can also be utilized. You only have to create a Python module to represent them in in AWW, and make this module activate the actual robot, for example via a system call.

When robots need to store data, they are free to create new tables in the database. Controlling this through AWW's interface would be difficult to facilitate, as the users would have to know the details of the robot's workings. However, at some point some robots will need input from the user. For now, this will be solved by setting variables in the source code of the robots.

4.3.5 Target URLs

In most cases robots will need information about which URLs to target. This information can be specified through parameters in the user interface of AWW. The URLs can also be loaded from file and assigned to a task. The database contains a table with names of tasks, and URLs they are meant to be used with them.

When a robot is activated, URLs can be passed along as arguments. If no parameters are passed to the robot, it can request the CCC to pass on any default URLs that are stored for that particular robot, or it can perform tasks that do not require any input-URLs.

Parameters

If a robot should be able to perform more than a single, clearly defined task, some additional parameters will usually be needed. Exactly which parameters is impossible to predict, and therefore difficult to include in the user interface. Python is dynamic, and allows functions to receive arguments of changing numbers and types, but the user interface is more rigid.

In this project the interface will not have support for additional parameters, and the problem must be solved in another way. For example, the robot designer can include a configuration file, and inform about it in the description of the robot. Alternatively, since Python is an interpreted language, configurations can also be modified in the source files themselves. That would mean that the most basic users will only be able to use the robots with the default configuration, performing one single task, but can enable more functionality with only a little learning required.

4.3.6 Scheduling

In the previous section we looked at grouping URLs, and assigning them to the various tasks. When we have such a structure in place, only a little work is required to enable scheduling in our program. Since we have frequency of execution in the information contained in our *tasks*, scheduling only requires starting a thread that monitors this information and activates the appropriate robots. Therefore this will also be attempted implemented.

If AWW is running in the default mode, the user can activate the scheduler manually, and leave the program running. In one of the other program modes, no user interface will open, but the scheduler will start automatically. Ideally the process could then automatically be set to run in the background.

If the internal scheduler should should turn out to be unstable, it is

possible to use an external one, and specify the command to be executed along with the start-up parameters for AWW.

Frequency of task execution will be described in a similar way as it is in the Crontab on Linux systems.

4.3.7 Program components

The program is structured after the *Model View Controller (MVC)* pattern. The CCC represents the controller. The views are represented by the command prompt and the GUI. The Model contains the database, which contains what is collected by the bots. It also contains all file handling functionality.

By default the program will start in GUI-mode, but by passing flags, or if a window manager is not detected, a command line mode may be opened instead. The GUI will show the command line, as well as other ways to access the functionality. The GUI has access to the controller directly, as well as through the command line.

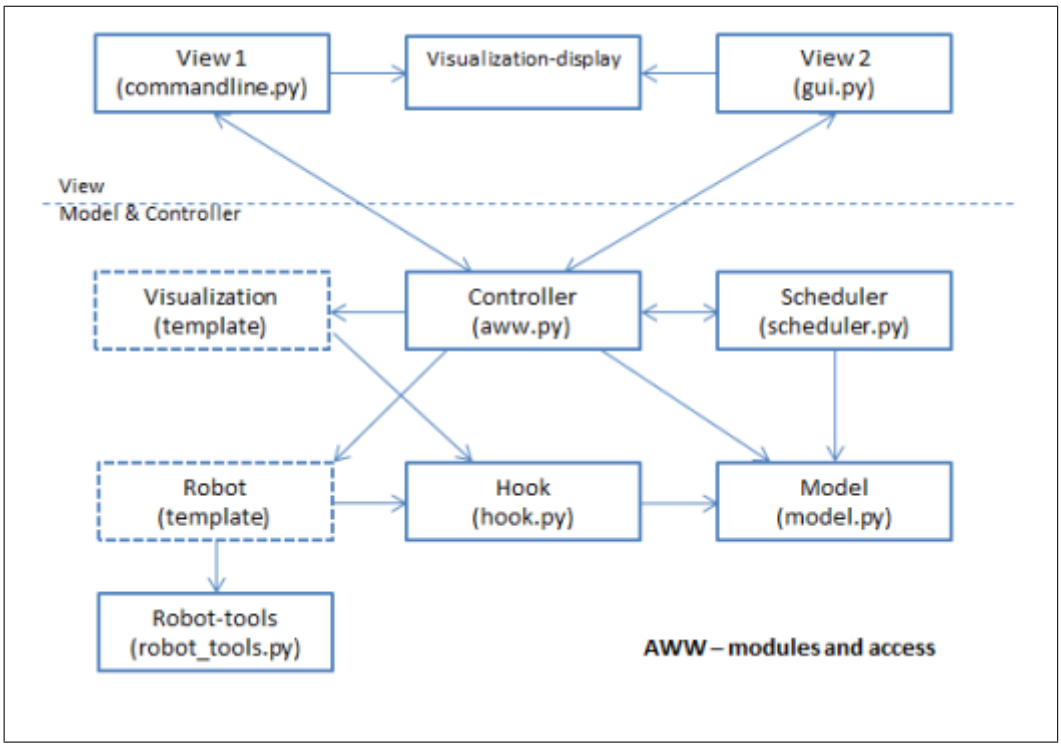

Figure 4.1: An overview of the modules that AWW is made of, as well as their intended access patterns.

Figure 4.1 on page 43 shows an overview of the program structure. The boxes represent Python modules, and the arrows specify access. Although

Python does not enforce access control, modules only make use of functions in other modules if this is indicated by the arrows in the figure.

Most of the program will be located in one folder. There will be two subfolders where users can add more functionality to the program, one for visualizations, and one for robots. Information about visualizations and robots will be stored in the database, when they are registered. On start-up the program goes through lists of robots and visualizations, and tries to register those that are not already present in the database.

Visualizations appear in the GUI, or in a spearate window if the GUI is not running. They can also be opened in the default browser, or exported to file as a raster image.

4.3.8 Commands

All the functionality of the program should be available through *one-liners*. This is done because it enables the design of graphical frontends to be similar to that of the command line. For example, if all access to program functionality can be summed up as single, complete statements, then they can easily be designed as forms on a web page.

Now comes descriptions of the various commands, sorted into groups of similar functionality. It is included in this document because it gives an overview of the programs functionality. Arguments to commands are marked with angle-brackets in this way: <argument>. In some places, arguments of commands are marked with a '+'. This also is also used in the help text at run time. It means that the syntax requires one or more the given argument. This is a type of notation commonly used in computer science, and expected to be understood by the type of users that AWW is intended for.

Command group - Robots

Commands:

- **bot list**
  list available robots
- **bot run_default <bot name>+**
  run robot, possibly with URLs from a task with frequency not set
- **bot run_url <bot name> <url>+**
  run a robot once, with specific URLs
- **bot run_with_task_urls <bot name> <task name>**
  run a robot once, with URLs belonging to given task

New robots are automatically imported on start-up. *Bot list* prints the robots that have been successfully imported. When activating a robot, it is possible to pass on a list of URLs, specify a task name for URL look-up, or run the robot without any arguments. There is a Python module called robot_tools, with commonly used functionality, and HTTP-request functions that monitors web traffic for the entire program.

Command group - Visualization

Commands:

- **viz list**
  list visualizations
- **viz <dataset name> <viz name>**
  open a visualization
- **viz browser <dataset name> <viz name>**
  open a visualization in the default browser
- **viz export <dataset name> <viz name>**
  export visualization in png format

The visualizations for this project will be made by the use of Excel, with exported data sets. So at the end of this project, these *viz* commands will not be meant to be used. They are only included to make it convenient to create extensions for visualization at a later time.

The *viz* commands can be run, but only to display a demo-visualization, that does not make use of any data sets. In addition to opening the visualization in AWW's GUI, it can also be opened in the system's default browser, in an external window, or written to disk.

Command group - Dataset

Commands:

- **set list**
  list datasets
- **set peek ¡set name¿**
  print 10 entries from a dataset
- **set export <file-format> <set name>+**
  export dataset to file (xml/html/txt/sql)

- **set truncate <set name>**
  delete all data in a set

The datasets are stored in the database during the execution of the web robots. The robots are free to create new datasets when it is needed. As a convention the sets should always have names that start with the name of the robot that uses it, followed by an underscore, and then something describing its contents.

Datasets are accessed by the visualizations through a programming hook. They can also be exported to several file formats, for further processing, or storage.

Command group - Tasks

Commands:

- **task list**
  list tasks

- **task run <task name>+**
  execute the command belonging to a task

- **task info <task name>**
  list URLs in a task

- **task create <task name> <command>**
  create a task

- **task frequency <task name> <minute hour dom month>**
  set frequency for execution of a task (you can use 'manual' and 'default')

- **task add_url <task name> <url>+**
  add a URL to a task

- **task import_urls <task name> <filename>**
  import URLs from file

- **task remove_url <task name> <url>+**
  remove a URL from a task

- **task remove_ task <task name>**
  delete a task, and its collection of URLs

Tasks are created manually. They contain a task name, a command, and an execution frequency. It is also possible to add URLs to a task, regardless of what command the task contains. If the task is to activate a robot, the URLs will be passed along to the robot. Adding URLs to a task can also be done by importing them from a file.

On creation no execution frequency is assigned to a tasks. It must be specified manually afterwards, in a syntax similar to that of *Cron*[2], with a syntax where minute, hour, day of month, month are typed as integers separated by a whitespace character.

Command group - Scheduler

Commands:

- **scheduler start**
  automatically iterate and execute tasks
- **scheduler stop**
  stop automatic execution of tasks

The scheduler will be started when the program is started is a certain mode. It can also be started from the user interfaces.

Command group - Miscellaneous

Commands:

- **gui**
  open the gui
- **help**
  display list of available commands
- **quit**
  exit the program (Ctrl-D)

Help prints a list of all commands, along with explanations for each command. *Gui* opens the graphical interface, and freezes the shell until it is closed again. *Quit* exists the command loop, and with no interfaces open, or the scheduler running, the main module will exit.

[2]Cron is a program available on Linux systems that enables users to schedule executions of commands.

4.3.9 Persistence

The robots may collect large amounts of data, so there is need for a database. This can be solved in many ways, but to simplify installation, the database will be stored as a file. The use of Object-relational mapping (ORM) has been considered. It would be convenient to specify classes that robots can instantiate and fill with collected data before passing them on to be stored by the model. This would make it convenient to control what datatypes are accepted for storage, as well as which additional parameters must be provided. In the end the library SQLite[3] was chosen for this project. It is available through the package Pysqlite[4], which is included in newer versions of the standard Python distribution.

This means objects are not used. The database is managed by SQL-statements in the Python code, but these statements are all placed in the module called Model, and the other modules are not affected by this choice.

> *SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world.*[20]

During execution the robots may insert into the database at any time. This is done by calling functions in the *hook* module, passing the data in the form of tuples (which is a datatype in Python). Control with which tables belong to which robots is not enforced, but the robots can ask for these to be created.

When requesting a table to be created, the robots supply their name, and an additional name as arguments. The table name is constructed by combining the two. The robot also provides names and descriptions for the columns in the table. In SQL it is originally possible to store column descriptions along with the tables. This functionality could not be found in the sqlite library, so column descriptions are not stored in the current version of the program.

There are some tables in the database used my the main module, but most of the tables are created by the robots, and referred to as datasets. Table 4.1 show and example of what a dataset can contain.

[3] `http://www.sqlite.org/`
[4] `http://www.pysqlite.org/`

timestamp	host	path	http_code
2013_02_19_15:40:22	www.uio.no	/tjenester/it /brukernavn-passord/ikke-logge-inn.html	202
2013_02_19_16:00:00	www.mn.uio.no.no	/ifi/tjenester/it /hjelp/latex/	404
2013_02_20_10:00:13	www.barejazz.no	/live.php	202

Table 4.1: The table shows contents of an SQL-table belonging to the test robot Awwhttp.

4.3.10 Files and directories

The program is built as a common Python package. The the folder structure and contents reflect the guidelines from the Python documentation. The program files can be run from any location. Program output is written in a separate folder. The most relevant folders of the Python package are the following:

- **AWW** This is the main folder of the package. In addition to the program code, it contains files and folders that are usually present in Python packages, such as a readme file, a license file, a setup file, source code for the documentation, etc.
- **AWW/aww** This folder holds the source code of the CCC.
- **AWW/aww/robots** This folder contains a subpackage, named robots. When new robots are created, their source files should be placed here. The folder also contains a file named robot_tools.py, with functionality that is useful for web robots.
- **AWW/aww/visualizations** This folder contains a subpackage, named visualizations. If visualizations are created, their source files should be placed here. The folder also contains a file named visualization_tools.py, with functionality that is useful when visualizing datasets.

Output folders

Output produced by the program is placed in a sub folder of the user's home directory, called *aww_data*. There lies the database file, *aww.db*, containing the program's configuration and all collected datasets. This is also the location of the log file, *aww_log.txt*, which is used for output that is considered

uninteresting for average users. If AWW is started in a limited mode, where only the scheduler is running, all output will be written to this file. There is also a sub-folder named *output*. When datasets are exported, they should be written to this folder.

4.3.11 Packages and modules

In .py-file in the AWW-package corresponds to a Python module. Below are descriptions of all the modules in AWW.

Modules in the package aww

The *aww*-package is the implementation of the CCC. For a better understanding of how the modules described below work together, refer to the overview in figure 4.1 on page 43.

- **aww** Passing this module to the Python interpreter will cause the *controller* module to be imported, and started.
- **commandline** A command line interface.
- **controller** This module represents the CCC. The user interfaces and the *scheduler* module accesses other program functionality through this module. The *controller* module, the *scheduler* module, and the *hook* module are the only modules that accesses the *model* module.
- **gui** The graphical user interface.
- **hook** An access point for extensions. The functions here call similarly named functions in the *controller* module.
- **model** Most of the access to disk is done via this module. It contains all the database functionality.
- **scheduler** Some functions here are called by the *controller* module. The *scheduler* module itself accesses functions in the *controller*, and in the *model*, to retrieve and execute tasks.

Modules in the sub-package robots

Except for the modules *robot_tools* and *upcheck_ext*, all the following modules are robots, that can be activated by the CCC.

- **findsitemap** This robot will download the robots.txt for all the URLs that are given. URLs marked with *sitemap* in these files are collected, and stored in the database.

- **httpcodes** This robot will be used for testing while implementing the software. It will collect and store the HTTP status codes, from requests made for all the URLs that are passed to it.

- **plaincrawl** *Plaincrawl* will be a basic web crawler. By default it will only extract URLs from web pages, but it can be imported by other robots, and they can provide their own functions for further processing of the web pages.

- **upcheck** This robot will make HTTP-requests for URLs, and store various information about the replies, concerning when the corresponding web pages were last updated. It also calculates hash values for the web pages, for comparison with earlier, or later downloads.

- **upcheck_ext** This module will be used for post-processing of content downloaded by the robot *Upcheck*. It will create an additional table in the database, that represents the data from *Upcheck* in a way that is more useful for export and visualization.

- **dublincrawl** This robot makes use of the robot *Plaincrawl*, but provides extra functionality, which extracts information about the presence of Dublin Core-metadata in the web pages.

- **robot_tools** This module is not a robot, but it contains functionality that is needed by several of the robots.

- **urlgen** *Urlgen* generates random IP-addresses, and use them to make HTTP-requests, in order to confirm whether the address points to a web server. If a reply is received, the address is stored.

- **url_list_refiner** This robot will use provided URLs for making HTTP-requests. When no error code or exception results, the address is stored in the database. The robot is meant as a tool to remove broken links from URL lists.

Modules in the sub-package visualizations

No visualizations will be developed in this project, but the system will be made to support that type of extensions too. A template is included.

- **surface** A template visualization. It will draw an example visualization, by using the Easyviz package.
- **visualization_tools** There will some functionality here, that may be useful when implementing visualizations.

4.3.12 Classes

There is little use of classes in AWW, but here are descriptions of a few of them.

AWW_GUI is a subclass of the *Frame* class from the *Tkinter* package. An efficient way to build a GUI is to subclass *Frame* and add the necessary widgets. In the case of AWW, the widgets are added to standard instances of the *Frame* class, which in turn are added to *AWW_GUI*. Starting the GUI means instantiating *AWW_GUI*. When it opens, standard *Frame* instances added to it cause visible borders to be drawn around different groups of functionality.

Dataset_info This class is defined in the module *robot_tools*. Its purpose is make robot creation more convenient, by letting the robots create one object of this class for each of their datasets, and fill them with all the information needed to create the corresponding tables in the database.

Robot_info This class is also defined in the module *robot_tools*. Instances of the *Dataset_info* class can be added to an instance of this class, along with other details about a robot, such as its name and description. This object can then be passed to the hook-module, supplying all the information necessary for registering the robot and its datasets.

PoliteOpenHandler HTTP-requests can be performed in Python through the use of various modules and functions. The module *urllib2*[5] defines the classes *OpenerDirector* and *BaseHandler*. By subclassing *BaseHandler* and passing it to an instance of *OpenDirector* it is possible to customize how HTTP-requests are made. AWW's *PoliteOpenHandler* class is such a subclass. If robots utilize functions available in AWW's *robot_tools* module the *PoliteOpenHandler* is used. In that way the correct user agent string is added to all HTTP-requests.

[5] `http://docs.python.org/2/library/urllib2.html`

Additionally, the *PoliteOpenHandler* ensures that global variables for traffic monitoring are updated, and that the program pauses execution for a given number of seconds if the network load is considered too high. If the network load is too high when execution resumes, the request fails and *None* is returned. The handler also makes use of the Python package *robotparser*[6] to determine whether requests can be allowed. If the parser returns *False* when requesting permission to retrieve a URL, the request is cancelled and *None* is returned to the robot that made the request.

One potential problem with the HTTP-functionality in the *robot_tools* module is that HTTP-redirects will be hidden. If a request is sent and a redirect code is received, the *OpenDirector* will follow the redirect, by making a new request, and return that result instead. This means that in robots where the identity of the received URL is important, care must be taken in choosing functions to use for making HTTP-requests. In some cases it may be necessary to implement individual functionality for the robot in question. However, if requests are made by robots, without going through the *robot_tools* module, the programmer should attempt to mirror the functionality of *PoliteOpenHandler*, to ensure that the correct user agent string is used, and to update global traffic info.

4.4 Robots

Here follows a detailed description of the main robots that are implemented, and used for data collection.

4.4.1 Main robots

2 main robots are created, and given the names Upcheck and Dublincrawl. Much of the functionality needed for Dublincrawl, is created as a separate robot, named Plaincrawl. Several other robots will be made, for example to aid in the creation of address collections to use as input for the data collection, but Upcheck and Dublincrawl are the robots that perform the actual data collection.

Robot: Upcheck

Upcheck's purpose is to find out whether web pages have changed. It does not use a crawler, but is handed a set of specific web page URLs. It has a much simpler functionality than Dublincrawl. All it does is to download

[6] `http://docs.python.org/2/library/robotparser.html`

pages from the web, and store information about them. Below follows a detailed description.

Name Upcheck

Purpose Determine whether web documents have changed or not.

Summary Takes one or more URLs as arguments, then downloads and computes an MD5-hash value. Stores the value, along with the last-changed value from the HTTP-header.

Scheduling Upcheck is meant to be scheduled, to describe how web pages change over time.

Arguments to main function page URLs (in our case sitemap addresses)

Execution time Average page download time multiplied by the number of URLs supplied.

Network load Depends on the number of URLs specified, and traffic settings in the CCC Only the URLs specified are downloaded, no URLs are added during execution.

Involved metadata standards None

Other standards HTTP

Data collected There are 3 datasets, named *start_urls*, *res_urls*, and *combined*. The last one is used by an additional module, named *upcheck_ext*, which is meant to be activated before exporting the collected data.

Table signatures:

- **start_urls** (rowid, timestamp, timestamp_excel, start_url)
- **res_urls** (rowid, fkey, seq_nr, received_url, reply_code, calculated_md5, header_md5, last_modified, expires)
- **combined** (rowid, fkey, timestamp_excel, start_url, seq_nr, received_url, reply_code, calculated_md5, header_md5, last_modified, expires, t_since_last_change)

Requirements Packages/modules: AWW
Anything else than Python: No

Modules upcheck, upcheck_ext

Robot: Dublincrawl

Dublincrawl makes use of the robot Plaincrawl, to crawl web sites. It supplies a custom function of its own, that counts the numbers and types of Dublin Core tags present in the web pages that are downloaded. Below are the details about the robot.

Name Dublincrawl

Purpose Determine degree and type of usage of Dublin Core tags in web pages.

Summary A number of URLs are given as arguments. Through the use of the crawler Plaincrawl, the corresponding sites are crawled. As the web pages are downloaded the numbers of different Dublin Core tags are recorded. When the crawling of a site is finished, a summary is stored in a dataset. When all the given sites have been crawled, values from counters for all encountered types of Dublin Core tags are transferred to another dataset.

A parameter named *crawl depth* influences how many pages that are downloaded. The URLs extracted from the web page corresponding to the starting URL represents the first level of crawl depth. The URLs acquired via those URLs represents the second level, and so on.

Scheduling This robot can be scheduled, but it is not its intended use. It is intended to create a snapshot of sites, more than investigating development over time.

Arguments to main function The function must receive at least one site. The robot will then perform crawling, and obtain new page-URLs on its own.

Execution time Depends on a parameter named *crawl depth*, as well as a limit on the number of pages crawled for each level. By default, 2 levels are crawled, with 30 pages on each level. Adding the seed URL, this means the execution time may be something like (1+(30x2)) multiplied by the average time required for downloading and parsing the web pages, and then multiplied with the number of web sites that are crawled.

Network load Dublincrawl is programmed for polite behavior, and does not exceed a set number of HTTP-requests per site, per minute. It does not follow inter-site URLs.

Involved metadata standards Dublin Core

Other standards HTTP, HTML, Robots exclusion protocol

Data collected Some data is stored in datasets belong to the robot Plaincrawl. Dublincrawl has 2 datasets, one containing summaries for individual web sites, and one containing total numbers of all Dublin Core tags found.

Table signatures:

- **summaries** (rowid, timestamp, timestamp_excel, site_url, page_count, dc_page_countdc_total, dc_coverage, dc_description, dc_type, dc_relation, dc_source, dc_subject, dc_title, dc_creator, dc_publisher, dc_rights, dc_date, dc_format, dc_identifier, dc_language)
- **tagsfound** (rowid, tag_name, count)

Requirements Packages/modules: AWW, BeautifulSoup, Lxml
Anything else than Python: The Lxml package has dependencies, that may require something in addition to Python.

Modules dublincrawl, plaincrawl

Dublincrawl gives a complex picture of an entire site. It is the most *bot-like* of the two main robots, in that it acts on its own and processes web content in a sophisticated way. It is not only an automated download of one specific web resource at specific intervals.

Robot: Plaincrawl

To provide crawling functionality for the robot Dublincrawl, another robot is created. It is named Plaincrawl. By default it simply crawls web sites, and stores discovered URLs in the database, but it can be passed Python functions via a *set*-function, in order to add processing of downloaded web content.

Name Plaincrawl

Purpose Crawls web sites, but as default includes no processing of the content.

Summary Plaincrawl crawls one or more web sites, that are specified on activation. The crawling is divided into levels, that correspond to the number of URLs that had do be followed to get from the starting URL

to some other URL. There is a limit on how many web pages will be downloaded on each level. A collection named *crawled* is used to prevent the same page from being downloaded multiple times, but this has no effect if different URLs point to the same web page.

Scheduling Execution can be scheduled, or started manually, depending on the type of datacollection.

Arguments to main function The same in Dublincrawl.

Execution time The same as in Dublincrawl.

Network load The same as in Dublincrawl.

Involved metadata standards None

Other standards HTTP, HTML, Robots exclusion protocol

Data collected All URLs discovered in the downloaded web pages will be stored temporarily, but when the crawling of a new web site is started, all the datasets are deleted, except for the one named summaries. During a crawl URLs are filtered, depending on whether they are judged to point to locations on the same site, or to somewhere else. The robots.txt for the site is also consulted, and may result in URLs being place in a dataset named *disallowed*.

Table signatures:

- **crawled** (rowid, url)
- **disallowed** (rowid, url)
- **outgoing** (rowid, url)
- **summaries** (rowid, site, crawl_depth, page_limit, disallowed, outgoing, crawled, level0, level1, level2, level3)

Requirements Packages/modules: AWW, BeautifulSoup, Lxml
Anything else than Python: The Lxml package has dependencies, that may require something in addition to Python.

Modules plaincrawl

4.4.2 Helper robots

A few robots will be made to assist the robots described above in their tasks, or to be used during the development.

Robot: Findsitemap The robot Upcheck, that was described in section 4.4.1, can be used with URLs pointing to many kinds of web content. In the data collection of for this project, it is used to monitor updates of sitemaps. In order to produce a collection of sitemap-URLs, another robot is created. Its name is Findsitemap.

Findsitemap works by downloading the robots.txt file corresponding to all the URLs that are passed to it. Robots.txt sometimes contains a *sitemap* attribute. The sitemap-URLs registered to this attribute is what Findsitemap collects and stores.

Allthough not all robots.txt files contain references to sitemaps, the ones that do often include several. Therefore this technique can potentially produce as many, or even more sitemap URLs, than the number of URLs passed to the robot.

Robot: Url_list_refiner The list of sitemap URLs that is generated by the robot Findsitemap is likely to contain broken links. Url_list_refiner is developed to make HTTP requests, and store the URLs for which a reply was received. The resulting set of URLs can then be used for a more time-efficient data collection.

Robot: Upcheck_ext Some processing of Upcheck's datasets will be required before they are exported, but should not occurr every time Upcheck is activated. A separate module is created for this purpose. It is not a robot in itself, but is implemented in a similar way, to give users access to it via AWW's interface.

Robot: Httpcodes A simple robot named Httpcodes is created. All it does, is to make HTTP requests for URLs that are passed to it, and register the HTTP reply codes. Httpcodes is used while developing the CCC, to help make it clear what functions are needed for communication between the framework and the robots.

Robot: Urlgen A robot by the name *Urlgen* is built with the purpose of generating random IP-addresses. The details around this are described in the implementation chapter. The word *URL* in the name of the robot can be misleading, but is something remaining from the early stage of the design, when it was hoped that it would be possible to obtain random URLs, instead of IP-addresses.

4.5 Visualizations

For this master project, visualizations will be produced in Excel, using exported versions of the datasets. Functionality for visualization was originally intended to be included in the program design. This task turned out to be too demanding for this project, but some of the functionality has been included. Throughout the text there are references to it, and in the user interface, options can be seen for running visualizations. However, it is not included in the goals of the project, and it will not be used for the data analysis.

The CCC will have functionality that lets visualizations be written to an image file, opened in the graphical user interface, in a separate window, or in the default browser. Opening in a browser means that the visualization is exported to an image file, which location is then passed to the system's default browser. No complete visualizations will be included with the program, but there will be a demonstration module, named *surface* that draws an example image, without making use of anything from the database.

4.6 Graphical user interface

Two solutions have been considered for a graphical user interface. One is to design both the GUI and visualizations to be displayed in a browser. The other is to use common Python libraries for graphics. Both options give a high degree of portability. Using Python means a large part of the system can be implemented using one single programming language, but it may require use of packages that are not included in all Python installations. Using a browser, on the other hand, is very convenient on the display side, as most systems provide a browser, but creating web pages with forms for the GUI may require installation of a PHP-server, or software with similar functionality.

Eventually, Python was chosen for the GUI, in line with my preferences, as well as to provide a simple installation, and to reduce the number of languages in use. Python will be used for as much as possible in the software. If possible the GUI and display output will be displayed in one and the same window. The python package used in *Tkinter*. Newer packages provide more functionality, but this one seems to be included in most standard distributions of Python.

As mentioned, visualizations can be displayed in a browser as well. That means that even if AWW runs in a shell without support for window management, it can be run in command line mode, and write the visualizations to disk, and then they can be displayed in a browser instead. This is a highly

portable solution.

Some rarely needed functionality will only be available through the command line, and not in the GUI. However, the command line is also displayed as a part of the GUI, so that functionality is still available. Aside from this both the command line and the GUI will have very similar functionality. The difference is mainly in how parameters are input. In the previous section about commands it is described how the commands are sorted in groups of similar functionality. For each of these groups, *bot, viz, etc.*, the GUI will contain a button. Next to the buttons are text fields and drop down menus to select options for the commands.

In this manner complete commands can be pieced together. When a button is pressed in the GUI the functions called in the controller module are exactly the same as those called from the command line. If, at a later time, someone decides to make web centric solution for the software, the same design can be used for a layout of forms on web pages. In this way, when a button is pressed the information from the text fields and drop down menus will be sufficient to build a command, which in turn can be passed to the controller via the system command line.

4.6.1 Output redirect

When the GUI is opened output for the whole program should be redirected to a text widget in that interface. That way output from the whole system can be printed in exactly the same way as before. This is convenient, especially in relation to output from the robots. The robots are ignorant of anything but the functions included in the program hook, and that module supplies no printing functionality. However, with output redirected, the robots can call on the standard print function, and the output will still be printed in the GUI.

4.7 Portability

The aim is for the software to have a high degree of portability. It should be possible to start it for anyone with a standard Python installation. Some of the Python dependencies, may not come with the standard installation. Robots and visualizations can make use of additional modules, so the number of dependencies is bound to grow along with the number of extensions. However, the main features of the program should always be available. The program will simply report that it is unable to start the GUI, or some robots or visualizations. Database support is the biggest challenge. It is based on

the *Pysqlite* package, which is only included in newer distributions of Python.

4.8 Extendability and API

Extending the functionality of AWW can be done by adding robots and visualizations. This is done through an API, as described below.

What robots and visualizations will be added later is difficult to predict, and so is the functionality they will require. This will be solved by letting the extensions decide everything on their own. The user interface will only support passing along URLs to robots. If more parameters or configuration is needed, this must be modified manually in the code files for the extensions. This means standard execution will be a simple procedure, but also that even the smallest change in configuration of extensions requires the user to have a basic understanding of Python as well as the program structure.

In the API extensions are meant to use three access points. There is a Python module called *hook*, which mostly provides functions for modifying or listing contents in the database. The folder containing robots has a module name *robot_tools*, and the folder with visualizations has the module *visualization_tools*. These two modules provide commonly used functionality for robots and visualizations, and they are not strongly connected to the main part of the program. All the modules, and most of their functions, are described in the user documentation for AWW.

In reality all the functions of AWW are accessible to any Python program that imports it. For example, a new type of user interface, or a browser plug-in, can import the module aww, and call any function in it. Other modules than the ones named above are not designed to be accessed directly, but there does not appear to be much access control in Python.

4.8.1 Importing extensions

When robots or visualizations are added, they should be copied to the folders with the same names. They can be written in any language supported by the operating system, but the link to the main program has to be through a Python module. The name of the module should be added to the list of extensions in the file __init__.py (which exists in both folders). The list are iterated every time AWW is started, and new entries are automatically added to the database.

When an extension is to be used, its Python module must be imported. This does not rely on a file path, but on the package hierarchy. For example, by importing *'robots.' + 'some_robot'* instead of *'robots/'+'some_robot'* one

can avoid challenges concerning path syntax, location of program files, or determining the current working directory.

4.8.2 Adding new robots

As described in the user documentation, creators of robots must include a set of functions for AWW to call. An example of how the code for a robot can look will be given in the implementation chapter, under the section *Boilerplate code for robots* (5.4).

4.9 Threads

There are many points in the execution where threads could be useful in this software: when the GUI is opened from the command line, when the scheduler starts a task, when a robot makes HTTP-requests to a list of URLs, during execution of any command with multiple robots or tasks as arguments.

A thread is created when the scheduler is started, but that is the only place threads are used in this project. Therefore the software may be slow and possibly fail under some conditions. For example, if two tasks are scheduled to execute at very close points in time, *starvation* may occur in the scheduler.

Another example is when the program pauses due to high traffic load. If the limit that was breached concerns numbers of HTTP-requests per minute, to one specific web site, we could still continue making requests to other sites. Instead, in the simple solution used here, the whole CCC will pause execution for several seconds.

4.10 Documentation

Program documentation and API, as well as the user documentation is distributed with the software. All the documentation can also be viewed online. The location is described in appendix A.

Additionally the graphical interface will have text labels that can be clicked, to open windows with information about the program, and how to use it.

Tooltips[7] do not seem to be available by default in the standard Python distributions, but some third party code has been found, that will be attempted added to buttons and checkboxes.

[7]Help text that is displayed when the mouse moves over a graphical component.

More information about commands can also be acquired through the command line. Examples of this are descriptions of robots, datasets, and visualizations. It would be useful if these descriptions could be displayed as graphical components, but the space in the program window is limited. If there was time to implement a more complex interface, different types of functionality could be split up into separate displays. That would make it possible to include more detailed information about each type of functionality.

4.11 Chapter summary

The design is finished. We now have a lot of details about how to implement the software. In the following chapter, this is put to use in the implementation and use of the software.

Chapter 5

Implementation

The software created for this project continues to grow, and at the time of writing the source code contains more than 4600 lines of code and comments. Initially the program was controlled via a command line interface, but when most of the functionality had been implemented, a graphical user interface was also added. Figure 5.1 on page 66 contains a screenshot of the graphical interface. It is programmed using an old Python graphics package, called *Tkinter*, which is included in most common distributions of Python.

5.1 Sourceforge project

Project name: Academic Web Watch

An account was registered at `http://sourceforge.net`, under the username Fivecode. This account was then used to create a project, named the Academic Web Watch. Sourceforge has provided several online tools. It has not been attempted to recruit other developers or users there, but services like version control, ticket management, project description, download support, and license selection has been useful.

5.1.1 Distribution

The software created in this project can be downloaded from Sourceforge at:

`http://sourceforge.net/p/wwatch`

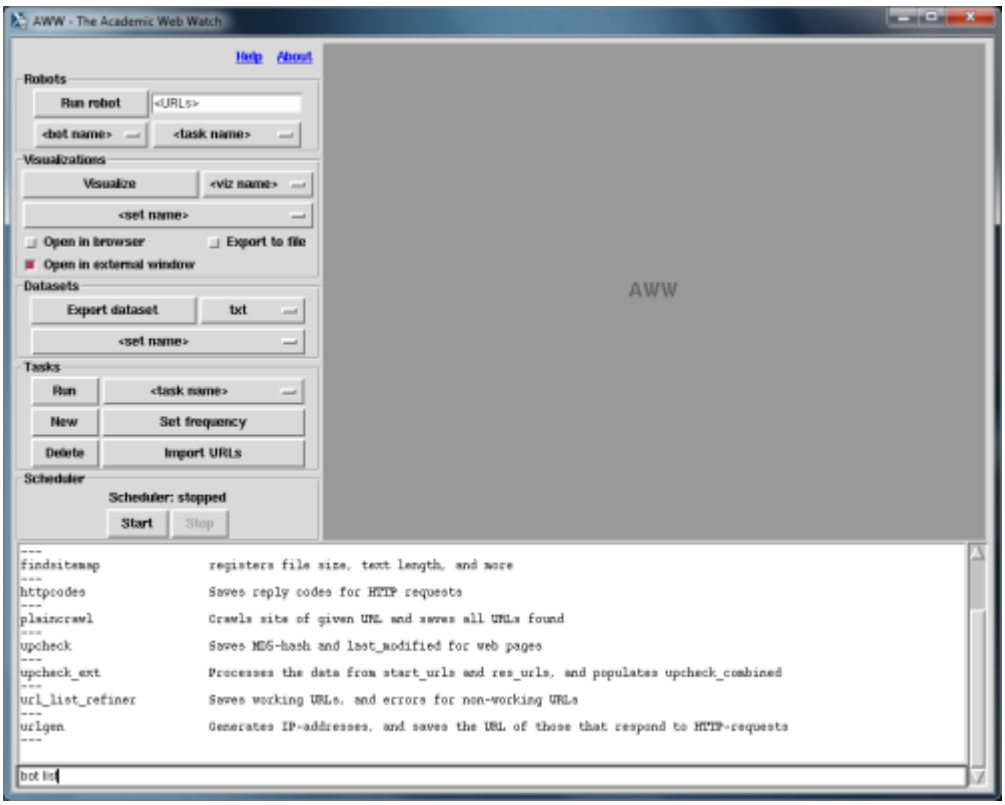

Figure 5.1: This is the graphical interface. The command line interface is included as a part of it, so that less frequently used functionality can still be accessed.

5.1.2 Download and installation

There are downloads for different versions of the program. They contain executable sourcecode structured as a Python package. One way to install the software would be to first acquire a working Python installation, then add the required modules described in the program documentation. After this the downloaded code can be executed from the commandline or imported while running Python. However, as it is structured as a Python package, it can also be added to the Python installation, making it available for import whenever Python is run, regardless of current working directory.

5.1.3 License

Soureforge allows their developers to chose between a selection of open source licenses for projects. AWW has been published under a *GNU Affero General*

Public License. The choice of license is meant to make the code available to as many developers as possible, while ensuring that anyone who starts a new branch of development must share their code. In short, the source code should be open, and stay open.

One of the things that sets the chosen GNU license apart from other alternatives, such as the Creative Commons licenses, is that it includes the feature *copyleft*. Copyleft specifies that that, not only distributed copies, but also modified versions of code must preserve the same rights as the original version.[1] This means that with copyleft other developers can make commercial branches of software, as long as the source code is still distributed along with the software.

In addition to the common features of GNU licenses, the Affero license has qualities that are meant to close a loophole related to web centric projects. For some previous GPL licenses, distribution of source code can be omitted when the software in question is a web service.

> *If your software can interact with users remotely through a computer network, you should also make sure that it provides a way for users to get its source. For example, if your program is a web application, its interface could display a "Source" link that leads users to an archive of the code.*[2]

Before people could make a web interface, and then be except from the requirement of distributing the source code, but as the above quote points out, that is not acceptable under the GNU Affero GPL.

Using Python also encourages open source code. The qualities of Python are suitable for open source projects. It is not common to distribute it without the source code, and the compiled code can usually be reverse-engineered.

Regardless of whether the software from this project is further developed, it is still hoped that it contributes something to the open source society, in that it limits what other developers can claim rights to, by being *previous art*. If someone else create software that works in the same way, they can not claim the design patterns as their own invention.

To label the project with the chosen license, two thing have been done. The license has been chosen, under the project settings on Sourceforge. Additionally a text file, named LICENSE.txt, has been added to the code repository. The file is added to comply with a description of Python package creation.[3]. The content of the file is a text describing the AGPL license,

[1] `http://www.gnu.org/copyleft/copyleft.html`

[2] `https://www.gnu.org/licenses/agpl.html`

[3] `http://guide.python-distribute.org/creation.html`

copied from `http://www.gnu.org/licenses/agpl-3.0.txt`, on the 19th of June, 2013. Afterwards, the name of the text file was then written as the value of the variable *license*, in the file *setup.py*.

The dependencies of AWW are controlled by various types of licenses. This does not affect AWW's license, because the dependencies are not distributed with the AWW Python package. Dependencies are listed, in the file *setup.py*, but must be acquired by the end user, as seems to be the standard for Python packages.

If a *bundled installer* is made for AWW, then some dependencies may have to be included. In that case the licenses for those dependencies must also be considered. If it is legal for the dependencies to be distributed along with AWW, then a copy of each license could be placed in a sub folder (e.g. the *docs* folder), and listed in an appropriate place.

An appropriate place is likely to be the *setup.py* file. Another common practice is to list dependencies and their licenses in a file named *manifest*. Files named *MANIFEST* and *MANIFEST.in* already exist in the program folder, but are used for other purposes, and may not be suited for inclusion of license descriptions.

Initially AWW was registered at Sourceforge with the Creative Commons Attribution License. Later, GNU Affero GPL was found to be a more suitable choice, and selected in the project settings. Due to technical problems on Sourceforge, this change was not reflected in the project's introduction page. Dialog with the staff makes it clear that this has to do with caching, and updates, and that it is not clear how long it will take before the information is displayed correctly.

5.2 Web address collections

5.2.1 IP address generation

Before the data collection can be initiated a collection of Internet addresses must be created. It is unrealistic, and maybe impossible to analyze the whole web, so a small sub-set will be used. It is desireable that the sub-set should be representative of the Internet as a whole, as that would make an analysis of collected data more valuable. Without representative data, it is impossible to draw any final conclusions based on the results.

Using randomly generated IP-addresses is a common approach to randomization of Internet addresses. The reason is that if there is no access to the whole set of possible addresses (URL-addresses or IP-addresses), it is

difficult to introduce randomness in any other way. An attempt was made to generate such a random collection. This turned out to be a more challenging task than first assumed.

A web robot *Urlgen* is created for this purpose. It is made on the same form as the other robots included with AWW, but ignores arguments to the start function. When activated it enters a loop that repeats 250 000 times. For each round in the loop 4 random numbers from 0 to 255 are generated, and used to build an IP-address.

The IP-address is then tested, to determined whether it describes a reserved address. If this it not the case, the string 'http://' is prepended to the address, and a HTTP-request is made for the header of whatever web page should happen to be located at the address. If the request succeeds, the address is stored in the database, along with the *Received-URL* attribute from the header. The *socket timeout* is set to 2 seconds. Therefore slow servers may be ignored, even though a working IP-address is used. When the address is stored, or the request has timed out, the end of the loop is reached.

During the experimentation with this robot several problems were encountered. One of them is automatic handling of redirects. The HTTP-request is made by the use of a function, *polite_get_header()*, provided among the functions available for all AWW-robots. It is preferred that those functions are used, because they also adapt to, and update the current network load. However, the function used here hides page redirects.

There are many factors that can be said to reduce the representability of this address collection. The low socket timeout may cause some valid addresses to be ignored. A server may be offline momentarily. Furthermore, there is not a one-to-one relation between IP-addresses and websites. Sometimes multiple IPs refer to one and the same web site. In addition, IP-addresses are likely to lead to certain types of web pages. They are more likely to lead to the main web pages of web sites, rather than their sub-pages.

Results of IP address generation

In the end the randomly generated collection was not found suited for use in the data collection. It is instead performed with a manually built URL collection. Therefore the results produced must not be viewed as representative for the Internet as a whole. It should still be possible to use them for analysis, as long as care is taken when generalizing over them. Aside from this, it should also be unproblematic to use the collection in a demonstration of how the software works.

Proc	Start time	End time	Generated	Reserved	Failed	Hits
1	18:06:22	07:46:45	250 000	34518	215465	17
2	18:06:25	07:28:26	250 000	34656	215340	4
3	18:06:29	07:31:56	250 000	34354	215646	0
4	18:06:32	07:42:33	250 000	34235	215764	1
5	18:06:35	07:53:18	250 000	34170	215816	14
6	18:06:39	07:37:58	250 000	34469	215531	0
7	18:06:42	07:40:49	250 000	34431	215568	1
8	18:06:46	07:50:09	250 000	34164	215829	7
9	18:06:49	07:48:47	250 000	34322	215675	3
10	18:06:52	07:38:45	250 000	34143	215841	16

Table 5.1: Information about the performance of the robot Urlgen, run as 10 separate processes.

5.2.2 Manually built address collection

A few general categories for web pages are described below. To obtain a broad selection, an attempt is made to collect URLs from all of those categories. Examples of categories are: geographic location, static and interactive websites, rarely updated and frequently updated web pages. Other categories of web pages can be found at Wikipedia's article about websites.[4] They also have a selection of lists of websites, including listing by popularity.[5]

Below is a list of URL sets that will be used, sorted by the names of the files they are stored in.

URL-collection files

urls_top100_mostpopular.txt
Amount: 100
The first 100 of *Today's Most Popular Web Sites*, collected on April 4th, 2013, from:
`http://mostpopularwebsites.net`

urls_bottom100_mostpopular.txt
Amount: 100
From the *Worst Performance within the Top 1,000,000 Websites*, collected

[4] `http://en.wikipedia.org/wiki/Website`
[5] `http://en.wikipedia.org/wiki/Lists\_of\_websites`

on April 4th, 2013, from:
`http://mostpopularwebsites.net/losers/1000000/`

urls_all.txt
All the collections, merged into one file.

urls_sitemaps.txt
In short, it is a collection of sitemap-URLs. The file urls_all.txt was used with the robot named *findsitemap*. This produced a collection of URLs that *findsitemap* extracted from the *sitemap*-attribute for robots.txt-files for the given URLs.

urls_top100_alexa.txt
Amount: 100
100 of the *top 500 sites on the web*, collected on April 4th, 2013, from:
`http://www.alexa.com/topsites`

urls_top100science_alexa.txt
Amount: 100
The first 100 of *Top Sites in:All Categories > Science*, collected on April 4th, 2013, from:
`http://www.alexa.com/topsites/category/Top/Science`

urls_countries.txt
Amount: 100
This collection has 4 URLs from each of the 25 top domain names corresponding to the countries listed in table 5.2 on page 72. The only criteria used, is that the countries should be spread out, and ideally with variation in quality of Internet service.[6] The names are chosen from a list on this web page: `http://www.computerhope.com/jargon/num/domains.htm`.

Most of the URLs have been found by searching Google with searches on the form: *site:.<domain name>* Some are also found by searching Wikipedia with a '.' followed by the domain name. Many of the URLs are site-addresses, which means a re-direct is likely, if sending a HTTP-request for the pages.

urls_misc_domains.txt
Amount: 50

[6]Quality here means difference in reply time, or other features that contributes to reliable and efficient Internet service.

Location	**Domain name**
Australia	au
Afghanistan	af
Argentina	ar
China	cn
Faroe Islands	fo
France	fr
Germany	de
Hong Kong	hk
Iceland	is
Morocco	ma
Montenegro	me
Mexico	mx
New Zealand	nz
North Korea	kp
Norway	no
Russia	ru
Somalia	so
South Africa	za
South Korea	kr
Spain	es
Sudan	sd
Sweden	se
Taiwan	tw
Trinidad and Tobago	tt
United Kingdom	co.uk
Zimbabwe	zw

Table 5.2: Names of a selection of countries and areas, along with corresponding top level domains.

This collection has 5 URLs from each of the 10 top domain names described below. Here we use other categories than country, and the names are chosen from a list on this web page:
`http://en.wikipedia.org/wiki/List_of_Internet_top-level_domains`

As with the country domain suffixes, most of the URLs have been found by searching Google with searches on the form: *site:.<domain name>*

The domains are:

net, org, edu, gov, mobi, int, travel, aero, pro, info

urls_newspapers.txt
Amount: 100
These URLs are collected from a list of *World Newspapers Online*, on 8th of May, 2013. They were found here:
`http://actualidad.com/`

urls_universities.txt
Amount: 100
These are university web sites, sampled on 8th of May, 2013, from this URL:
`http://www.shanghairanking.com/ARWU2012.html`

5.3 Data collection

With the contruction of the address collections complete, the data collection can be started. Data collection is described in the method chapter, but since the robots had not been designed, there were few details available. Here follows a detailed explanation of how the robots are executed in order to collect material for visualization and analysis.

5.3.1 Automatic data collection with the robot Upcheck

The first attempts

For a larger part of the duration of this project, one or more configurations of AWW has been set up to run automatically. For most of that time they did not generate useful output. There have been difficulties in getting the program to run in a stable manner. Processes have been terminated by the operating system, when taking to long to finish, or running with too high priority. At times most of the http requests made failed, for various reasons.

Other times everything appeared to run as planned, but nothing was written to the database.

Final setup of automatic execution

Upcheck was scheduled to run twice per day, starting from the 16th of September, 2013. At this point the system was stable enough that, even though problems arise, it should be possible to see from the output what kind of events have occurred.

The data used for analysis, and for producing visualizations, was exported on the 30th of September, but the software was allowed to run until the 25th of October, 2013.

5.3.2 Manual execution of the robot Dublincrawl

The robot Dublincrawl was activated manually, instead of through scheduled activation. The visualizations are based on one single execution, that was done on the 27th of October, 2013.

Dublincrawl uses the largest URL collection, referred to as *urls_all.txt* in section 5.2.2. This URL collection contains over 1000 web addresses. In addition, 30 addresses are added, from a web site that lists them as using Dublin Core tags.[7] If the average use of Dublin Core tags on the web is low, the latter should ensure that there is still some useful output. The robot had the parameter *crawl depth* set to 2, but in order to complete the execution in a reasonable time it was only allowed to crawl 20 URLs on each level of this *crawl depth*.

5.4 Boilerplate code for robots

Here is the code from a test robot for AWW. The code is included here, because is an efficient demonstration of how AWW works, and because it is useful for creation of new robots. To create a new robot, simply copy this code, and modify it to perform new tasks.

The functions aww_register and aww_run(hosts=None) are the only functions that must be present in all AWW's robots.

```
#!/usr/bin/env python
```

[7] `http://trends.builtwith.com/feeds/Dublin-Core` (visited on 16th of September, 2013)

```
"""
.. module::httpcodes
   :synopsis: This module's purpose is to save reply codes
   for HTTP requests.

   When aww_run is called a HTTP request is made for each
   of the URls that are supplied as arguments. The HTTP
   status code of the reply is stored to the database.

.. moduleauthor:: fivecode <fivecode@users.sourceforge.net>
"""

# import some standard python packages
import time, datetime
import urllib2, hashlib
import hook
# import some of aww's functionality for web robots
from robot_tools import Robot_info, Dataset_info, HttpError
from robot_tools import get_timestamp, make_well_formed
from robot_tools import polite_open, get_standard_fields

bot_name = 'httpcodes'

def retrieve_code(url):
    """
    Obtain and store a HTTP reply code for the URL given as
    parameter.

    If the HTTP request results in a redirect, an exception
    will be raised. This is just a test bot, and the
    redirect will not be followed

    Of the 3 functions in this robot, this is the only
    one that is not required in all robots for AWW.

    :param url: A complete URL to use for a HTTP request
    """
    try:
        file1 = polite_open(url)
        if not file1: # open failed
            return None
```

```
        result = get_timestamp(), url, file1.getcode()
        file1.close()
        rowid = hook.dataset_insert(bot_name, \
                                    'results', \
                                    result)
    except HttpError, e:
        if e.http_code == 301 or e.http_code == 302 or \
            e.http_code == 303 or e.http_code == 307:
            print bot_name +' received redirect'
            print 'This test robot does not follow redirects.'
        else:
            print 'HttpError ({0}) received: {1}'.\
                format(e.http_code, e.args)
    except Exception, e:
        print bot_name + ' received exception: ' + str(e)

def aww_run(hosts=None):
    """
    A function with this signature must be present in all
    robots for AWW.

    This function may be different in different
    robots, but should always handle the hosts-argument
    for the cases None, not None, and list

    :param hosts: URLs that can be used by this robot.
    """
    if hosts is None:
        hosts = hook.get_default_urls(bot_name)
        if hosts is None:
            print 'Could not acquire host URLs'
            return
    if type(hosts) is not list:
        aww_run([hosts]) # convert hosts to a list
    else:
        for url in hosts: # for all URLs given
            retrieve_code(make_well_formed(url))

def aww_bot_register():
    """
    A function with this signature must be present in all
```

```
    robots for AWW.

    Its purpose is to register the robots name,
    description, and datasets in the main framework.
    """
    bot_info = Robot_info(bot_name, \
                              'Stores HTTP reply codes')
    # SET 1
    set1 = Dataset_info('results', '(rowid, timestamp, '
                        + 'url, reply_code)')
    set1.add_field('timestamp', 'text', \
                       'Time of execution')
    set1.add_field('start_url', 'text', \
                       'URL provided by the user')
    set1.add_field('reply_code', 'text', \
                       'Reply code from HTTP header')
    bot_info.add_dataset(set1)
    hook.bot_register(bot_info)
```

5.5 Revisions

The scheduled execution of Upcheck was performed with version 0.0.9 of the software. The robot Dublincrawl used version 0.1.0. The last changes done to the software in this project resulted in version 0.1.1. All these distributions are available at the Sourceforge project.

5.6 Program dependencies

It has been attempted to chose well established, widely used software libraries. For example, because the old graphics package, Tkinter, has been used for implementing the GUI, it is likely to be usable on most common Python distributions. Concerning persistence, the package pysqlite is a newer addition to the standard Python distribution, but also in widespread use.

When a system does not support a part of AWW's functionality, it is hoped that AWW will fail gracefully. In other words, remaining functionality should still be useable. For example, if GUI libraries are not available, command line mode should be started instead. If individual dependencies for robots or visualizations are not available, other robots and visualizations

should not be affected. Lastly, if the program runs on a system unable to display graphics, it should still be possible to write visualizations to disk.

Additionally, all dependencies for main program functionality are Python packages. Generally, Python packages can be installed to a user's home directory, without any need for special privileges. Sometimes these dependencies will have other dependencies, and rely on non-Python software. If that software is not installed on the system, it may be a problem. For example, the Python package *easyviz* is crucial for visualization. In theory it should be possible for the user to install it, but if none of the backends for *easyviz* are available (GNU-plot, Matplotlib, etc.), then installation will require special user privileges.

The dependencies for the CCC are listed in a file called *setup.py*, as this seems common for Python packages. An average Python installation is likely to be missing some of these dependencies, but they can be added, for example by the use of one of Pythons package managers. Most of these dependencies can be installed on a linux home directory without requiring any extra user privileges.

The dependencies for the robots must be handled individually.

5.7 Chapter summary

That concludes the implementation chapter, and the development of the software. The chapter described the creation of a project on `sourceforge.net`, details on how the program design was implemented, and how the data collection is performed. In the next chapter the result of the data collection and the outcome of the project will be discussed.

Chapter 6

Findings

Now follows descriptions and discussion of the output from the data collection. After that the project outcome and the completion of the research objectives will be discussed, resulting in a conclusion for the whole project.

6.1 Output from the robot Upcheck

The visualizations for Upcheck are based on a data collection period of 2 weeks, during which the robot was scheduled to run twice per day. The software collected data for more than a month, as described in section 5.3.1, but the visualizations are based on data from the first 2 weeks of that period. This is partly because there was no more time left for working with the data, but also because the resulting table for those 2 weeks in itself resulted in a table containing 40076 entries. Importing the data for the entire period into Excel could be difficult without further processing of the dataset.

6.1.1 Data collection results

Initially the robot Upcheck stores its results in two separate tables. An extension is then run, which combines the results in a third table. This table has the following columns:

- **rowid** An automatically included index present in all tables
- **fkey** An integer describing the location of the entry with the start URL in the table *start_urls*
- **timestamp_excel** The time at which a sitemap download occurred

- **start_url** The URL that was used for the initiating request (which are sometimes followed by redirects)
- **seq_nr** In the case of redirects, a sequence number is used to describe the order of resulting table entries. In this table only the final HTTP reply is included, but the sequence number shows how many redirects occurred.
- **received_url** The URL of the downloaded document (When redirects occur, this differs from the start URL.)
- **reply_code** The HTTP reply code of the last request
- **calculated_md5** An MD5 value for the download, calculated by Upcheck
- **header_md5** An MD5 value extrated from the HTTP header
- **last_modified** The last-modified value from the HTTP header
- **expires** Expiry date/time from the HTTP header
- **t_of_previous_change** A timestamp describing when a different MD5 value was last calculated for this start URL
- **change_interval** If the MD5 value calculated is different from the previous entry, then it is calculate how long the previous value lasted

The column *start_url* describes URLs that were passed to the robot from the CCC. When the robot is activated each such URL may result in multiple entries in the dataset, if redirects should occur, but the same timestamp and start URL are used for all of these entries. When we look at update frequencies of web pages, the start URL will be used as a label. If a URL is redirected to a different location between two subsequent program executions, this will be viewed as if a page has been updated - except if the MD5-values are exactly the same.

Post-processing For the analysis in this project only a few of the columns are used, but there is another important reason for creating the robot extension that outputs the table. It allows for post-processing to be performed on the data. This way two variables can be calculated here, by the use of Python, instead of in Excel, which would be more complex.

The two values calculated are *t_of_previous_change* and *change_interval*. The are created in an iteration of the whole combined dataset. *t_of_previous_change*

is set to 0 for the first entry. For following entries the value is updated whenever a URL is noticed to result in a different MD5. This means each entry in the set has a value in this column, describing when a URL was last modified. This value is used to calculate values for the column *change_interval*, which describes the duration of time between two updates. During the iteration, when a URL is discovered to have changed, the timestamp in *t_of_previous_change* for the previous entry is subtracted from the new timestamp, that is generated for the current entry. The result is the duration of the duration the URL was in its previous state - in other words, the *change_interval*.

6.1.2 Visualization results

On the following pages are figures describing the output from the robot *Upcheck*. Most of them concern results for single URLs, and they give information about something referred to as *update intervals*. During the execution of the robot, if a URL is found to have been updated, this marks the end of one such update interval.

In figure 6.1 on page 82 the number of updates (or update intervals) found is reported for a small number of URLs.

As can be seen, the number of intervals varies greatly, which is another way to say that the change frequency for different URLs varies greatly. Another way of describing change frequency is to compute the average time between updates. This I will call the average interval length. Figure 6.2 on page 83 displays this information, again only for a handful of URLs.

The standard deviation from the average update intervals can be seen in figure 6.3 on page 84.

Figure 6.4 on page 85 contrasts the minimum and maximum update interval length for a selection of URLs.

Figure 6.5 on page on page 88 may be the most useful visualization, because it displays trends that describe the whole dataset, instead of just a selection of URLs. We can see that there is significant amount of URLs that have average update intervals with a length between 0.04 and 0.05 days.

6.2 Output from the robot Dublincrawl

6.2.1 Data collection results

After completing the crawl of a website, Dublincrawl transfers its findings to the dataset *dublincrawl_summaries*. Several of the columns in the table

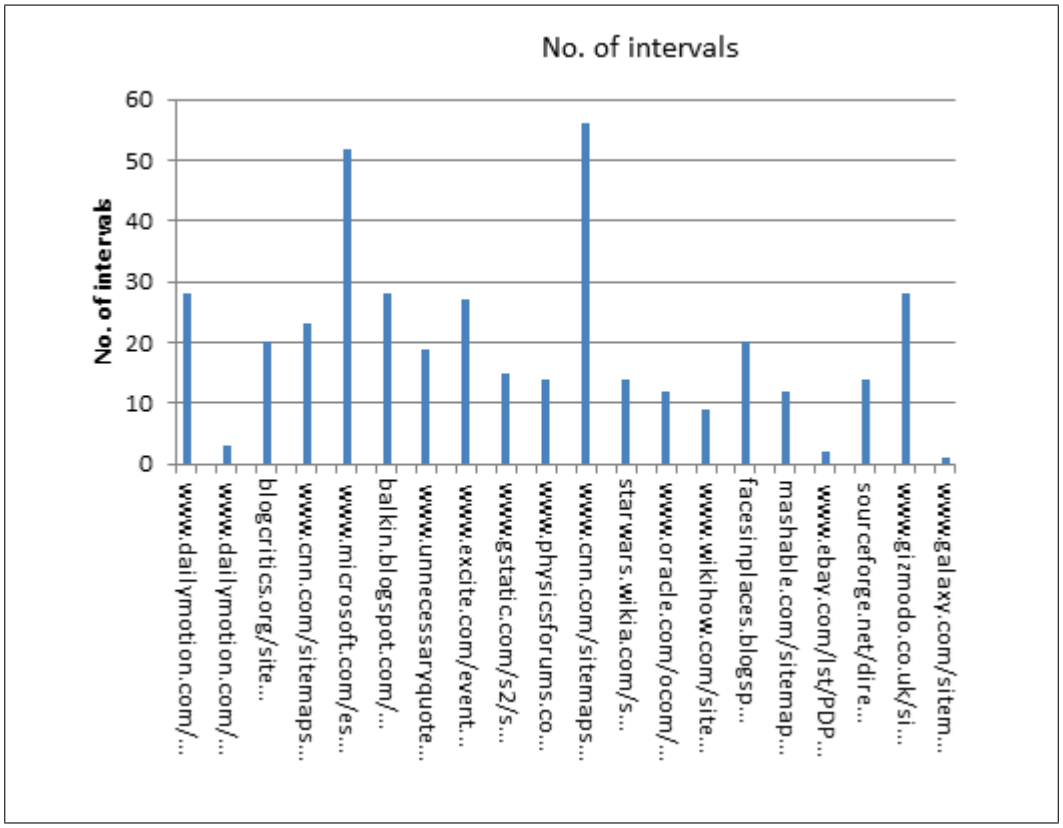

Figure 6.1: Numbers of updates (or update intervals) for a selection of URLs during two weeks.

contain numbers of tags found, for the most common Dublin Core tags. The selection of tags was done based on a list from `dublincore.org`.[1]

- **rowid** An automatically included index present in all tables
- **timestamp** The time at which a crawl ended
- **timestamp_excel** The timestamp converted to Excel's time format
- **site_url** The URL of the site that was crawled
- **page_count** Total number of pages crawled on one web site
- **dc_page_count** Total number of crawled pages where Dublin Core tags were found

[1] `http://dublincore.org/documents/2001/04/12/usageguide/generic.shtml` (visited on the 29th of October, 2013)

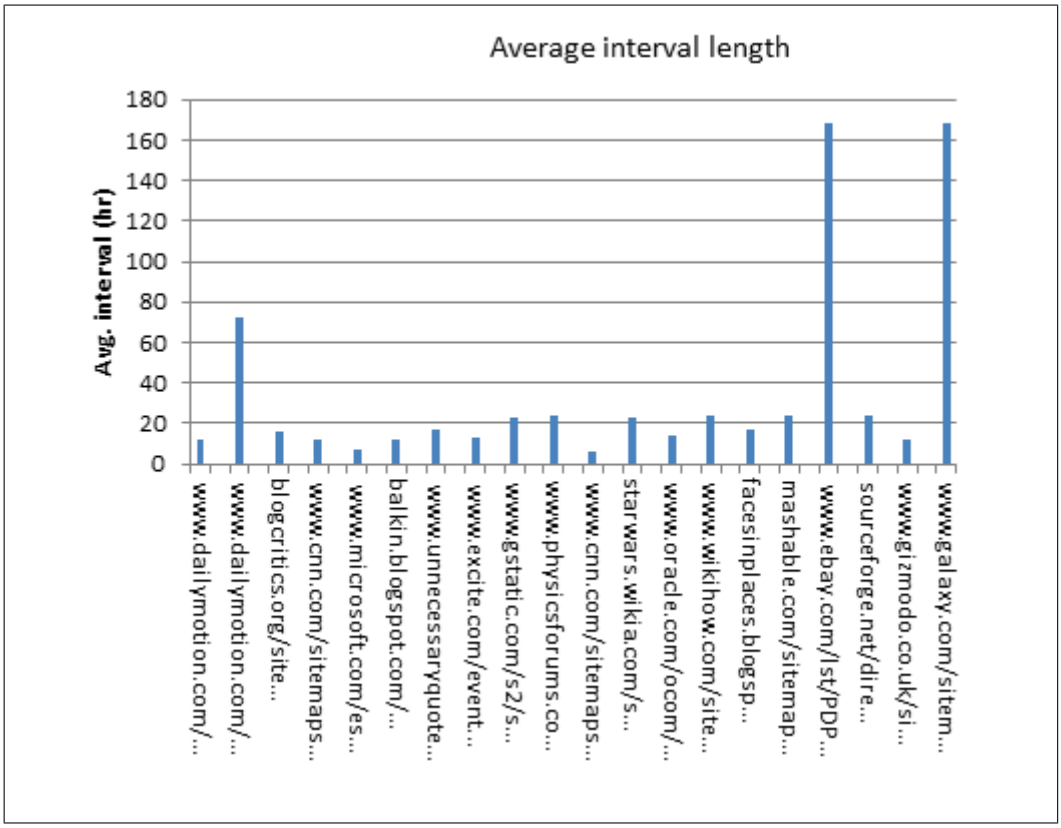

Figure 6.2: The average length of update intervals for a selection of URLs.

- **dc_total** Total number of Dublin Core tags found on a web site
- **dc_coverage** A type of Dublin Core tag
- **dc_description** A type of Dublin Core tag
- **dc_type** A type of Dublin Core tag
- **dc_relation** A type of Dublin Core tag
- **dc_source** A type of Dublin Core tag
- **dc_subject** A type of Dublin Core tag
- **dc_title** A type of Dublin Core tag
- **dc_creator** A type of Dublin Core tag
- **dc_publisher** A type of Dublin Core tag

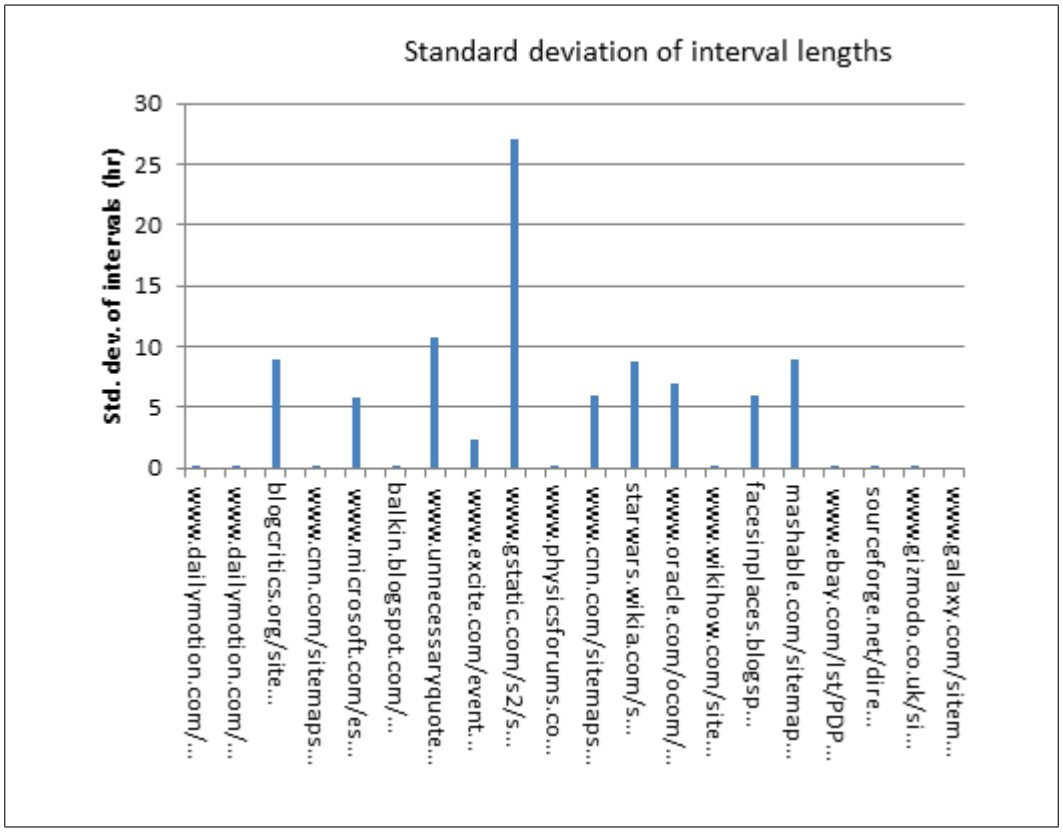

Figure 6.3: Standard deviation in the length of update intervals for a selection of URLs.

- **dc_rights** A type of Dublin Core tag
- **dc_date** A type of Dublin Core tag
- **dc_format** A type of Dublin Core tag
- **dc_identifier** A type of Dublin Core tag
- **dc_language** A type of Dublin Core tag

Dublincrawl also inserts data into another table, named *dublincrawl_tagsfound*. This table contains all categories of Dublin Core tags found, as well as the number that was found of each. The table contains the following columns:

- **rowid** An automatically included index present in all tables
- **tag_name** Names of all the types of Dublin Core tags found
- **count** The total number of tags found of the individual tags

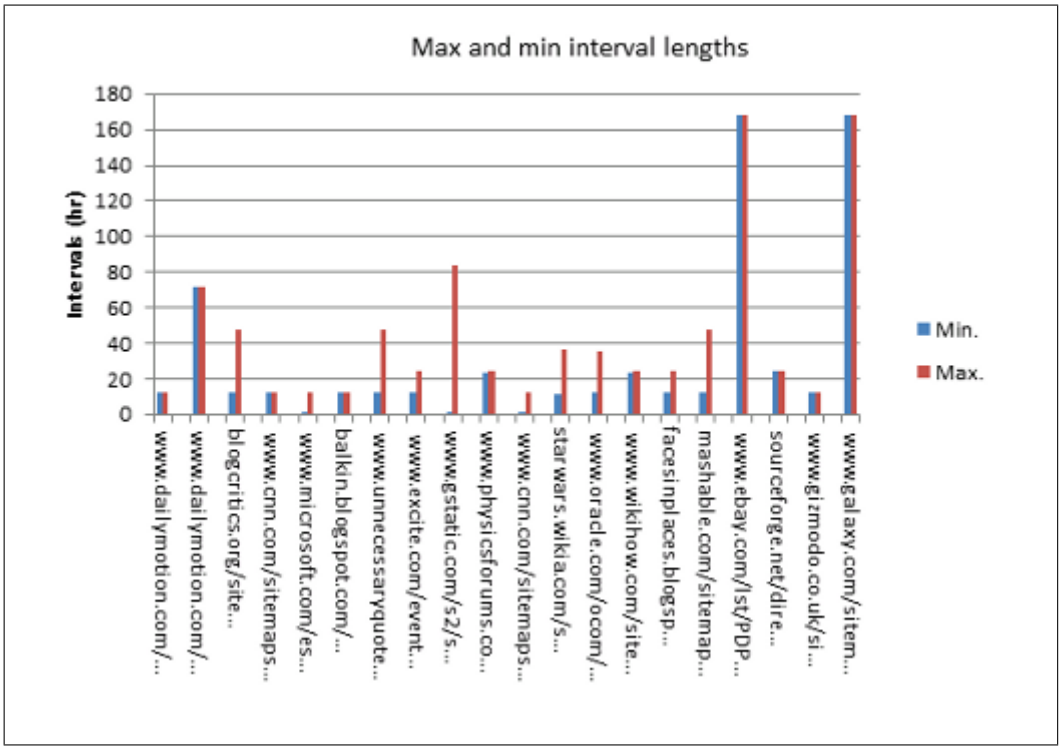

Figure 6.4: The longest and shortest update intervals for a selection of URLs.

Observed Dublin Core tags

Table 6.1 on page 86 shows the sites with most DC tags. Table 6.2 on page 87 shows the names of all the Dublin Core tags found.

6.2.2 Visualization results

Figure 6.6 on page 89 shows the number of Dublin Core tags per site. A total of 1027 sites were crawled, 52 sites had some Dublin Core tags, 975 had none.

Figure 6.7 on page 90 shows the Dublin Core tags that were found to be used most frequently.

Site	**Pages with DC tags**	**Average DC tags per DC page**
nhs.uk	70.5 %	13.0
weather.gov	90.2 %	8.4
canada.com	44.3 %	16.0
census.gov	96.7 %	5.9
loc.gov	59.0 %	7.5
dailykos.com	83.6 %	5.0
www.milblogging.com	26.2 %	11.0
regis.edu	25.9 %	11.0
store.usps.com	47.5 %	5.0
cdc.gov	78.7 %	1.8
diamond.jp	54.1 %	2.0
www.memepool.com	96.7 %	1.0
hotelsone.com	90.2 %	1.0
ers.usda.gov	30.0 %	4.0
www.washingtonpost.com	39.3 %	1.4
resumegenius.com	20.0 %	4.0

Table 6.1: The sites with most Dublin Core tags, found by the robot Dublin-crawl.

tag name	count		tag name	count
dc.title	431		dc.subject.epappt	1
dc.language	412		dc.subject.epasubstance	1
dc.subject	383		dc.subject.epahealth	1
dc.description	317		dc.robots	1
dc.creator	279		dc.date.disposal.review	1
dc.publisher	200		dc.format.medium	1
dc.rights	187		dc.subject.epacat	1
dc.date.created	161		dc.publisher.address	1
dc.identifier	144		dc.date.archivedate	1
dc.format	125		dc.coverage.x	1
dc.type	119		dc.subject.epachannel	1
dc.date.issued	109		dc.date.issue	1
dc.coverage	94		dc.date.x-metadatalastmodified	1
dc.date.reviewed	62		dc.creator.personalname	1
dc.date.modified	59		dc.subject.eparat	1
dc.contributor	58		dc.lmusitename	1
dc.date	48		dc.subject.epaopt	1
dc.source	45		dc.coverage.y	1
dc.keywords	44		dc.subject.epaect	1
dc.date.valid	27		dc.subject.eparit	1
dc.relation.ispartof	16		dc.subject.eparegulation	1
dc.coverage.spatial	15		dc.subject.epaindustry	1
dc.author	15		dc.rating	1
dc.unused	14		dc.subject.epaemt	1
dc.coverage.temporal	4		dc.description.abstract	1
dc.robot	3		dc.rights.copyright	1
dc.distribution	3		dc.subject.epabrm	1
dc.audience	2		dc.title.alternative	1

Table 6.2: Names of all Dublin Core tags found by the robot Dublincrawl.

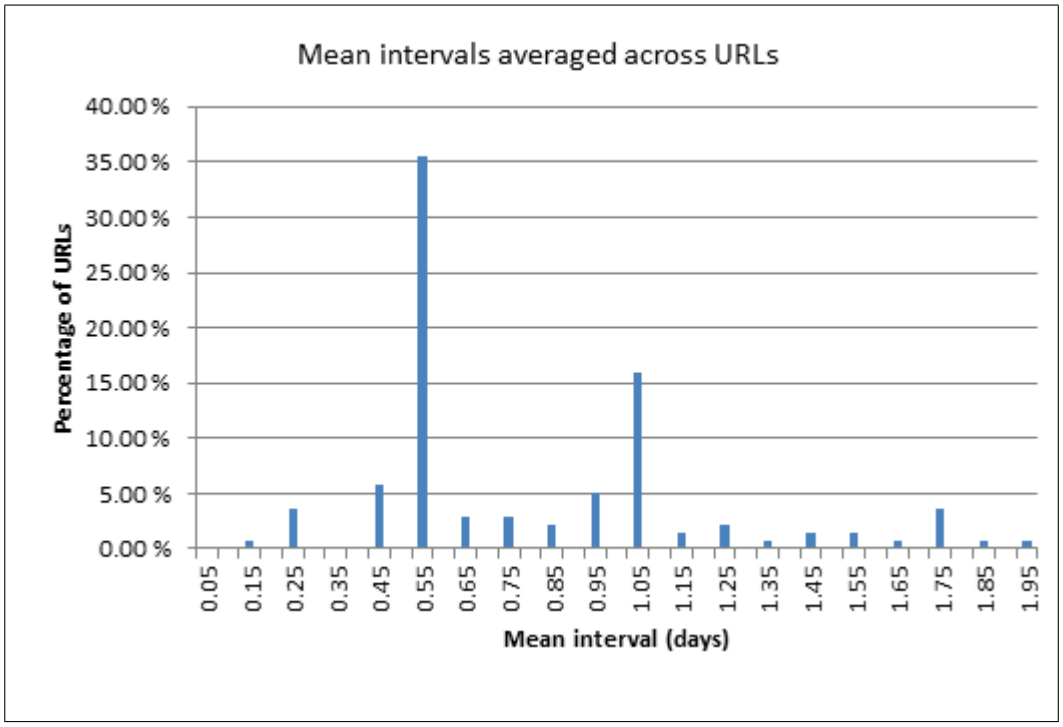

Figure 6.5: Mean lengths of update intervals per URL. The horizontal axis shows the mean intervals between updates, and the vertical axis shows the percentage of URLs that have this update interval.

6.3 Discussion

6.3.1 Choice of programming language

The choice of Python as the language for implementation has mostly been a positive experience. It was not based on previous experience with Python. I had never used Python before this project. The choice was instead based on online descriptions, and praise of Python by programmers at the University of Oslo.

One master project later much has been learned. It turns out that many problems can be quickly solved in Python. The user community is active online, and good advice has been easily available from sites like `http://stackoverflow.com` [2]

However, Python's standard libraries are sometimes not as complete as hoped. For example, there are multiple packages with functionality for the

[2] Questions asked in relation to the AWW project:
`http://stackoverflow.com/questions/16759035/python-variables-discord-across-modules`
`http://stackoverflow.com/questions/17127306/row-pop-function-in-pysqlite`

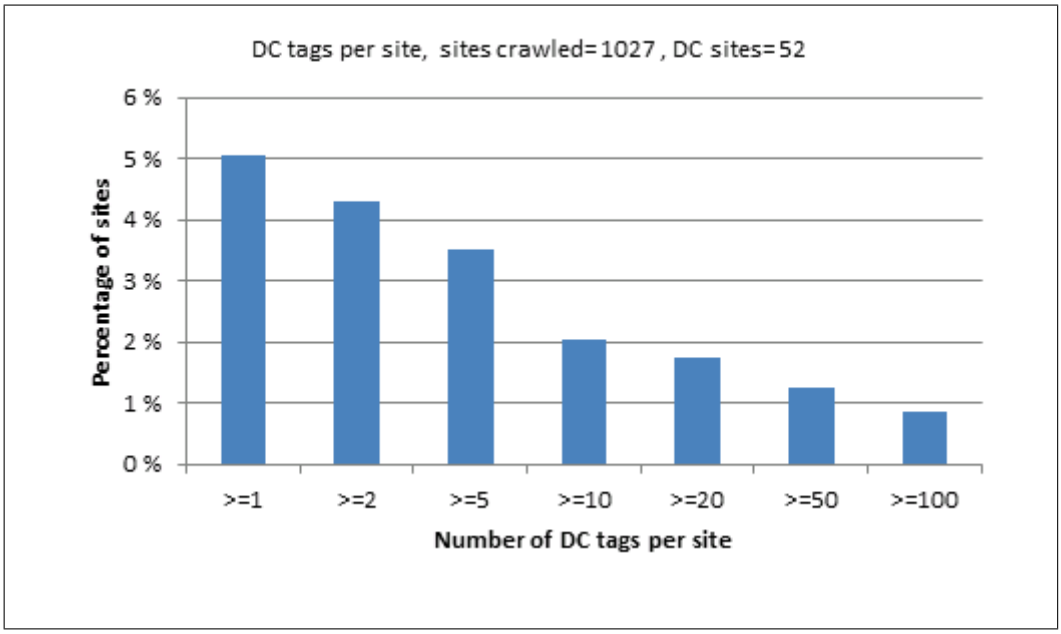

Figure 6.6: Number of Dublin Core tags per site.

web, but their implementations change significantly between Python versions, and they have overlapping functionality implemented through different algorithms. Likewise, the *Pyslqlite* library works well for common purposes, but some *sqlite*-functions, like *sqlite3_last_insert_rowid()*, do not appear to have been implemented in the Python bindings. Furthermore, documentation is often available, but is structured more freely, than for example Java's APIs. This means getting an overview of packages, modules, and functions can sometimes be challenging.

Exceptions from 3rd party code, for example from Python's robots.txt parser, have also been hard to debug. At times data collection has not functioned at all, and the only clues to go by were short error messages that can not be understood without knowledge of the internal workings of 3rd party libraries.

To sum up, despite some complications, Python has been found to be a good choice for this type of project.

6.3.2 The open source experience

Working in an open source environment has been fairly unproblematic. Open source software is often very straight forward, and does what is claimed, without any more formality than necessary. 5 minutes can be enough to

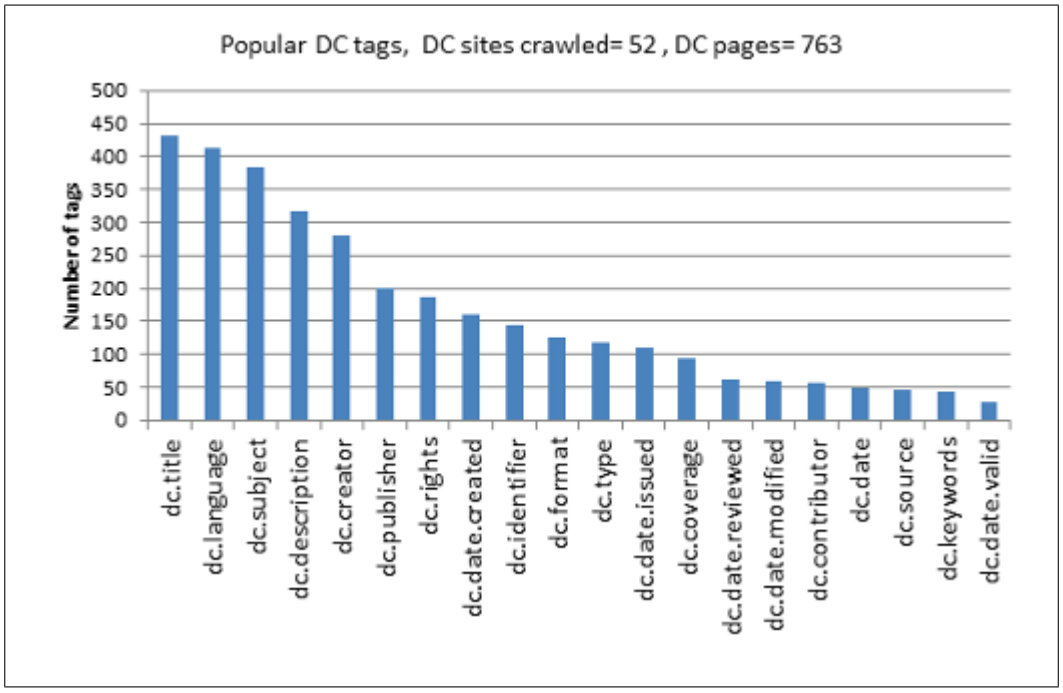

Figure 6.7: The most popular Dublin Core Tags found, extracted from 52 sites and 763 pages.

register a project on Sourceforge,[3] and fill in the required details. Then one can simply start coding.

In retrospect, it would have been interesting to try GitHub,[4] or maybe Google Code,[5] instead of Sourceforge. Still, Sourceforge provides many of the most common development tools. For example, you can chose between using Subversion (SVN) or Git[6] for version control.

Developing software under an open source license has created no problems. Then again, that is more of an issue for a commercial project. However, after this project, I would be interested in participating in a commercial project, under an open source license. Open source appears to mean simply that anyone can take up competition with your product, and competition could be a healthy influence, especially when it comes to ensuring quality in software.

[3] `http://sourceforge.net`
[4] `http://github.com/`
[5] `http://code.google.com/`
[6] `http://git-scm.com/`

6.3.3 Unforeseen challenges

Hidden redirects

There are many ways of downloading web content with Python. In the robots sub-package functions are written, that provide an uniform way for AWW's robots to do this, which also add a user-agent string, and update variables for traffic monitoring. This was implemented through the use of an instance of the *OpenDirector* class from the Python library urllib2. This is an object to which *handlers* can be added.

The default *handler* automatically follows redirects. After a custom redirect-handler sub-class was added, redirects are now handled by raising exceptions, to which robots may respond in individual ways.

Problems with the robotparser package

The use of the *robotparser* module, which is included in the Python installation, has been problematic. This module is meant to assist in the parsing of robots.txt files. If a URL is listed as disallowed in the robots.txt file, it means that robots should not follow that URL, but during the develpment of the robot Plaincrawl, several sites were crawled without a single URL being registered as disallowed. When the data collection with the robot Dublincrawl was done, many URLs were registered as disallowed, but it is uncertain whether the correct ones were registered.

Online discussions were found about the quality of the *robotparser* module, stating that *sometimes it seems to 'work' and sometimes it seems to fail.* [7]

This might be the reason why *Nikita the spider* was developed with its own parser for robots.txt files.[8] A similar solution could be considered for AWW.

6.3.4 Usability

AWW is a young program, and has a good deal of unintended behavior. Little emphasis has been put on the user experience, and the interface can be challenging to understand. The command line interface has been used more than the GUI, and should be manageable to most users that have a general understanding of what the program does. Although the GUI works, it needs to be developed further before it can be said to add to the usability of

[7] `http://stackoverflow.com/questions/15344253/robotparser-doesnt-seem-to-parse-correctly` (visited on the 30th of October, 2013)

[8] `http://nikitathespider.com/python/rerp/` (visited on the 30th of October, 2013)

the program. At present, although the GUI requires less typing, the choices in the menus are difficult to understand for new users.

The program functionality is easy to extend, but only for those who have thorough knowledge of how the program is structured.

6.3.5 Usefulness

This project has attempted to investigate how well web robots are suited for use in web analytics. We have managed to collect a significant amount of data, and were able to produce meaningful statistics with it. It appears that both the use of web robots, and, specifically the use of the software developed in this project, can be a powerful tool for analyzing the web.

On the other hand, a significant amount of work still remains before the software will be an efficient tool for research. The program is highly versatile - it can be modified to perform a very wide range of tasks online, but it can be challenging to configure for those who don't know it well.

6.3.6 Shortcomings

Towards the end of the project most of the goals have been realized. A program has been made that provides most of the prescribed functionality. Still, there are many features that could have been implemented in better ways, as well as problems for which no solutions were found.

Due to lack of time no user testing has been conducted. The program reached a usable state too late in the master thesis period for it to be promoted online, and users recruited. Feedback from other users would give a more thorough evaluation of AWW.

Using the software for more meaningful analysis of the web, with more specific goals, and a better URL collection, would also increase the value of this work. If visualization of the results was also supported by the software, it would be a more complete tool for ananlysis. For these too, the problem is the same. Namely that more work on the software would be needed, as well as time to plan and conduct further experiments.

I have learned from this project that programming can be a time-consuming task and many more months were spent writing the code than originally estimated. I also experienced that some of the Python libraries, that I expected to work, were less stable than desirable. When exceptions are raised by these libraries, the explanations they contain can be difficult to understand for someone without knowledge of the internal workings of the libraries.

Due to the previous, the outcome of the project is experimental software, that currently is only usable by those who have thorough knowledge of it.

Since AWW now is an online open source project, and a program with a wide range of functionality, I would suggest that any further development is done in smaller steps. For example, programmers or students could be asigned tasks concerning the functionality of individual modules.

Dataset descriptions When robots are registered they supply a short description of themselves, and of all the fields in their datasets. SQL is said to support storing of such information in a database, but this was not found possible with the *Pysqlite* library that AWW is implemented in. As a partial solution names of datasets, and their descriptions are stored in a dedicated table. The descriptions of fields, on the other hand, are lost.

More functionality for handling datasets Only entire datasets can be exported. Viewing or exporting data from specific time intervals would be useful.

Exceptions raised by 3rd party code Exceptions are raised and caught several places in the code, but it is difficult to interpret or predict exceptions raised by 3rd party code. For this reason such exceptions are often simply caught and discarded, after which the execution resumes in the next place that is assumed to be safe.

An example of this is exceptions resulting from HTTP requests. If the exception is the result of a redirect, with an error code of 301, 302, 303, or 307, then the exception is passed on for the individual robots to handle. If the exception is of another type, None is returned instead, and the exception is written to the command line or to a log file.

Visualizations Visualization internally in the program would be very useful. Although several required features for this are in place, it is not operational on any useful level. The remaining work required to make it this work properly could be enough to make up another master project.

GUI design The graphical user interface is hard to understand for users who have not had an introduction to the program. All the functionality is visible at all times, giving a large number of items that the user must relate too. Some functionality can still only be reached through the command line. For example, descriptions of robots can only be shown by typing the command *bot list*.

Graphics libraries The choice of Python's Tkinter library for building the GUI created some difficulties. No support was found for tabbed panels, and it was difficult to add tooltips. Furthermore, it was inconvenient to display generated images in the GUI. The images had to be saved to file, then converted from PNG to the GIF format, and finally reloaded. At some point it may become relevant to display HTML code in the GUI, in relation to visualization, but this too does not seem convenient to do by the use of Tkinter. Although Tkinter is portable, other libraries, like Wxpython may have solutions for all of the above problems.

Scitools and its Easyviz package, on the other hand, appear very powerful, portable, and simple to use. If the graphics produced by Easyviz can be drawn, for example to the *Canvas* class of Wxpython, then the combination of Scitools and Wxpython would be sufficient for many types of tasks that AWW can be designed to perform.

Tooltips The GUI was intended to include tooltips. Support was not found for this in Python's TKinter library, so third party code was used to implement it instead. That was later removed due to uncertainty around whether it was appropriate to use the code. Figure 6.8 on page 94 shows how some of the tooltips looked in version 0.0.9 of AWW. The origins of the third party code is described in the module *tooltips.py* of that distribution. The module can be re-integrated if this seems appropriate to future developers.

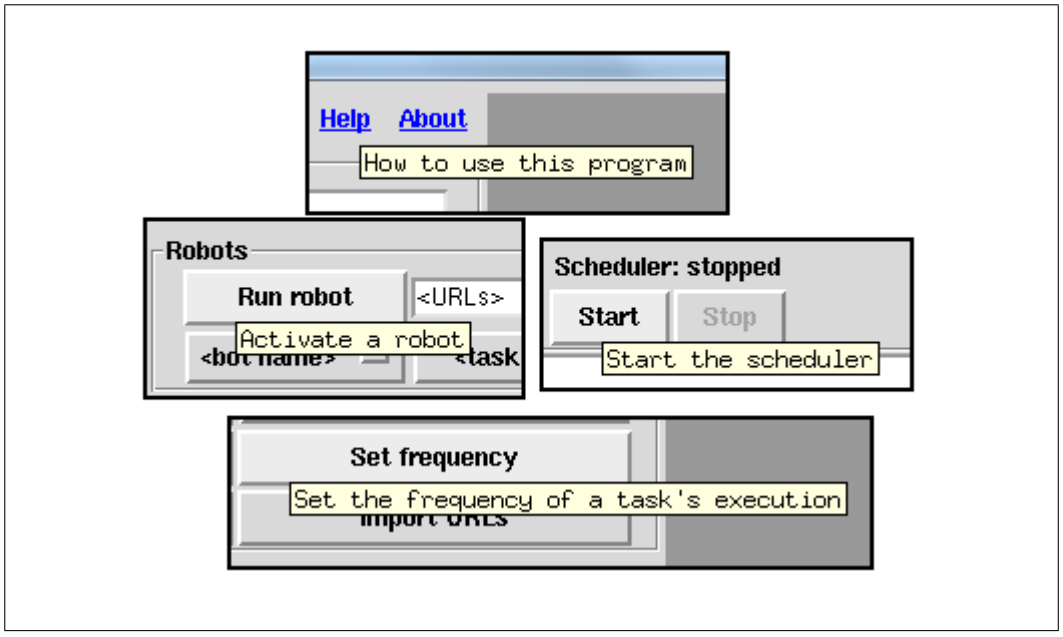

Figure 6.8: A few of the tooltips that could be seen in version 0.0.9.

Command parsing All the commands listed for the command line interface work, but the autocompletion functionality can be improved.

6.4 Conclusion

6.4.1 Research objectives

The research goals will now be considered again, to evaluate whether they have been realized.

Study the field (R1)

> *Study the field of web analytics and web robots, and summarize what exists.*

In the background chapter (chapter 2) an introduction was given to web analytics, the semantic web, and robots for web analysis. In relation to the semantic web, several standards for metadata were introduced, in order to explore what type of data web robots can be programmed to extract. In section 2.3, about web robots, descriptions were given of different types of web robots, as well as examples of their architecture and variations in implementation.

Although there are several references to academic texts, many of the references are to more short-lived, and un-evaluated online documents. This is largely due to the nature of the topics discussed. For some of them, online discussions is a more available source of information, than academic material, since program designers often write blogs, or user documentation, instead of academic articles.

Conduct experimental design (R2)

> *Conduct experimental design by defining and creating a Command and Control Center (CCC), a Graphical User Interface (GUI), and a couple of sample robots.*

In this project, the amount of time invested in programming may well exceed that which was needed to write the dissertation.

The choice of designing the software in an experimental way has been a good one. Without experience in creating this specific type of program, or prior experience with the programming language used, creating a more complete design before starting implementation would most likely result in

poor design choices. During the process several unforeseen issues appeared. For example similar functionality in the robots led to redundancy in the code. This was later solved by creating a common toolset in a separate Python module, called *robot_tools*. Incidentally, this could later be used by the robots to monitor the network activity, in a way that the total network load is visible to all robots. This is not very relevant if the robots do not execute simultaneously. If the use of threads is further developed they may well operate at the same time, but at that point a new challenge appears - namely that of racing conditions, and the possibility of starvation the traffic control functionality. These are all issues that were not considered in the original design.

Some of the functionality in the CCC was a prerequisite to building the robots. Its construction came to require a lot of time, and so less time was left for work on the robots. To those who are familiar with the system, creation of new robots can be done in a short amount of time. Had there been more time left in this project, it would have been very rewarding to try out various types of robots, and further develop the support they rely on in the CCC. In early distributions of the software, there are included several python modules which are beginnings of new robots, with a wide range of functionality. Developing this further would have been an efficient way to test and improve the system.

The research goal concerning experimental design is considered as having been met. The end product has much of the desired functionality, but is not yet suitable for average users.

Conduct experiments (R3)

> *Conduct experiments with the sample robots, and produce data and visualizations.*

Data has been collected from the Internet through the use of the robots named Urlgen, Findsitemap, Upcheck, Plaincrawl, Dublincrawl. A URL collection has been filtered through the use of the robot Url_list_refiner. Two datasets were processed to create a third, by the use of the robot Upcheck_ext.

The robot Upcheck has run for several weeks, creating a timeseries, while the robot Dublincrawl was used spontaneously, to investigate the state of web sites at the current time. Both robots received URL lists with hundreds of URLs as input, and performed large numbers of HTTP requests to collect data.

The data collected from both Upcheck and Dublincrawl has been used to generate several visualizations in Microsoft Excel. This was done more

as a demonstration of the software, than to test theories about information representation on the Internet, but there has been a discussion of the results, explaining how the visualizations display various characteristics of the web sites that they are represent.

Analyze the findings (R4)

> *Analyze the findings from the previous activity, and discuss the usefulness of robots in web analytics.*

The previous tasks produced several forms of output, which was evaluated in the current chapter (chapter 6 - Findings).

Discussion of the visualizations produced shows that the software can be used to examine features of web pages in a way that is valuable in scientific contexts, and for web analytics.

Aside from the output, the project has also given experience with designing robots for web analytics. It is felt that robots has potential for use in web analytics. At the same time, the project shows that the creation of custom robots for analysis can require a large amount resources. If a future project can continue to build on the software produced in this project, that will hopefully reduce the amount of work needed to achieve results.

Custom robots for web analytics can perform a very wide variety of tasks, and they may have applications, for which a cookie based approach is not practical. For example, robots give easier access to web sites owned by other people than yourself, because the way they function does not require any modifications of content on the web sites.

6.4.2 The project as a whole

The research objectives have been gone over. To sum up, the objectives have been achieved. A software program has been created, according to the specification, and has been used for data collection and analysis.

Although development has taken more time than estimated, the finished product was shown to be useful for data collection on the Internet. The visualizations display usable information about how web pages change over time, and how Dublin Core metadata is made use of in a selection of web sites. This indicates that web robots can be a valueable tool in the field of web analytics.

6.5 Chapter summary

In this chapter the output of the project has been described, and then discussed. First the output produced by the data collection was presented, and next the visualizations generated from this output.

All this was then used for discussion. The discussion partly concerns how the work has progressed, as well as observations about the Internet in general, that have been made through the software produced. However, the discussion's main purpose is to provide an evaluation of the software itself. This is accomplished by discussing its usability, usefulness, and whether or not it has enabled realization of the research goals formulated for the project.

At the end of the chapter the discussion draws to a close in a short conclusion, stating that all the main objectives have been achieved.

There is one final chapter remaining of this text. In it some comments will be given, about the quality of the work performed, and about the future of the software project on Sourceforge.

Chapter 7

Final word

7.1 Quality

In the previous chapter (section 6.3) there is a discussion of the outcome of this project. To sum up, the research goals have been met, and several features of the desired software have been implemented. Still, as the software became more complex, I also experienced some unintended behavior.

It has been demonstrated that it is possible to use the software for the collection of data that can be used for research purposes. However, it has also become clear that making a tool usable by average users will require more work. As it is now, the software is most useful to *expert users* - users who know the software's architecture well enough to modify it, and who can figure out the cause, when something unexpected occurs.

7.2 Future Work

Although some analysis has been performed in this project, the main focus has been put on implementing the software. The visualization and analysis that has been performed is useful for demonstrating possible applications for the software, but a more thorough analysis of trends in metadata usage on the Internet will require more resources, such as time for data collection, as well as further development of the web robots. Now that some of the groundwork is out of the way, it would be interesting to see if other master projects, or perhaps a research project can use the resulting software for more extensive analysis of metadata.

7.2.1 Desirable features

Precision in dataset export When datasets grow, it is useful to be able to work with subsets of them. Dataset export could therefore be improved by allowing parameters to the export functions, for both the time period to use, and for which table columns to export.

Perhaps a query generator could be created, for even more control over the database. A panel in the GUI could provide functionality for assisting the user in generating Pysqlite-queries, which can then be applied to the database. This would not only improve the export functionality, but make it more convenient to maintain, and update the various datasets.

Database queries could also be used to give robots and visualizations more control over their datasets, eliminating the need for specific Python functions to be created for every type of access.

Should query functionality be created, the use of query language should be given some consideration. Pysqlite has some limitations. Creating an abstraction layer, and for example using pure SQL instead, would make it easier to replace the database at a later time.

Improved scheduler The scheduler presently included in the software allows repetition of tasks at given intervals, but the specification of these intervals is could be improved. It works well for for specific points in time, or for one execution per minute/hour/day, but executions at given intervals specified in minutes or hours requires multiple tasks to be created. The time format used for scheduling is inspired by the *Cron* tool on Linux systems. It would be useful if this was fully implemented.

Threads More use of threads in the CCC would be useful. For example, threads could be used when robots are activated. Presently, users must wait for the robots to complete, before they can continue to use the interface.

If more threading should be used, it is important to consider implications for the control functionality relating to network load, in the module *Robot_tools*. If multiple robots are downloading at once, they will be using the same counters for traffic load. Racing conditions could cause some robots to terminate their execution, if the traffic load is too high for too long.

Dynamic parameters for execution of robots It should be possible to pass more parameters for the robots through the user interface. At an optional list of URLs can be passed on, but to keep the structure of the command strings simple, all other parameters must be set in the source code of the robots.

An example where the current conditions are impractical is the robot Plaincrawl. It contains settings for crawl depth, as well as a limit on the number of pages to crawl on each level. The robot is meant to be re-used by other robots, but setting these values in the source code of Plaincrawl itself would mean that all the robots utilizing it would use the same parameters.

The problem is currently solved by letting the individual robots call *set* functions in plaincrawl, before they start crawling web sites, but it would be a cleaner solution if any kind of parameter could be passed from the user interface. This would allow any robot to be used for multiple tasks, with multiple configurations, without requiring its source code to be altered between executions.

Visualization As mentioned in the discussion, creating useful functionality for visualization internally in the program could require enough work for another master project to be initialized. Should such a project be initialized, one of the first things that should be done is to reconsider the selection of graphics libraries to use. I believe that scitools and easyviz is one of the strongest candidates for providing what is needed for the actual visualizations. But integration of visualizations into the user interface, as well as any redsign of the user interface would likely benefit from replacing the Tkinter package with something more up to date.

For instance, by default Scitools produces PNG files as output, and Tkinter does not support PNG. The package Wxpython, on the other hand, can handle PNG files. Should there be need for it, it can also work with OpenGL content.[1] If it is desirable to output some visualizations as HTML tables, Wxpython can also render HTML in the GUI.

At present the GUI is very hard to understand for anyone without prior knowledge of the program. Wxpython can be used to split the GUI into tabs, containing different categories of functionality, thus reducing the number of items visible in the interface. Wxpython also has native support for *tooltips*.[2]

7.3 Chapter summary

We are now at the end of this chapter, and of the dissertation. In the chapter a few final words have been given, about the quality of the work that has been done, as well as the quality of the resulting software. Suggestions were also given for topics, that could be relevant to those who might wish to continue developing the software, or who work on similar projects.

[1] `http://www.wxpython.org/docs/api/`

[2] `http://wiki.wxpython.org/wxGrid%20ToolTips`

Bibliography

[1] Abello, J., Pardalos, P. M., and Resende, M. G. *Handbook of massive data sets*, vol. 4. Kluwer Academic Publisher, 2002.

[2] Abelson, H., Adida, B., Linksvayer, M., and Yergler, N. ccrel: The creative commons rights expression language. `http://wiki.creativecommons.org/images/d/d6/Ccrel-1.0.pdf`, 2008. [Online; accessed 14-May-2013].

[3] Adida, B., and Birbeck, M. Rdfa primer. `http://www.w3.org/2006/07/SWD/RDFa/primer/`, 2008. [Online; accessed 14-May-2013].

[4] Berners-Lee, T., Hendler, J., and Lassila, O. The semantic web. *Scientific american* (2001).

[5] Čapek, K. *RUR (Rossum's universal robots)*. Penguin. com, 2004.

[6] Cho, J., and Garcia-molina, H. Parallel crawlers. *Proceedings of the 11th international conference on World Wide Web* (2002), 124–135.

[7] Dimon, G. Microformats Primer. `http://www.digital-web.com/articles/microformats_primer/`, 2005. [Online; accessed 14-May-2013].

[8] Guinchard, C. Dublin core use in libraries: a survey. *OCLC Systems & Services 18*, 1 (2002), 40–50.

[9] Hemenway, K., and Calishain, T. *Spidering Hacks*. O'Reilly, 2004.

[10] Hendler, J. Agents and the semantic web. *IEEE Intelligent Systems Journal 16* (2001), 30–37.

[11] Heydon, A., and Najork, M. Mercator: A scalable, extensible web crawler. *World Wide Web 2*, 4 (1999), 219–229.

[12] Ihm, S., and Pai, V. Towards understanding modern web traffic. *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (2011), 295–312.

[13] Kristol, D. M. Http cookies: Standards, privacy, and politics. *ACM Trans. Internet Technol. 1*, 2 (Nov. 2001), 151–198.

[14] Kumar, D. K. S. Web crawler: A review. *International Journal of Computer Science and Managment Studies 12* (2012), 401–405.

[15] Langtangen, H. P., and Ring, J. H. Easyviz documentation v1.0. `http://home.simula.no/~hpl/easyviz/easyviz_sphinx_html/tmp_easyviz.html`. Accessed: 21/05/2012.

[16] Millett, L. I., Friedman, B., and Felten, E. Cookies and web browser design: toward realizing informed consent online. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2001), CHI '01, ACM, pp. 46–52.

[17] Mohr, G., Stack, M., Rnitovic, I., Avery, D., and Kimpton, M. Introduction to heritrix. In *4th International Web Archiving Workshop* (2004).

[18] Phippen, A., Sheppard, L., and Furnell, S. A practical evaluation of web analytics. *Internet Research 14*, 4 (2004), 284–293.

[19] Quboa, Q. K., and Saraee, M. A state-of-the-art survey on semantic web mining. *Intelligent Information Management 5*, 01 (2013), 10–17.

[20] Sqlite.org. Sqlite. `http://www.sqlite.org/`. Accessed: 30/05/2012.

[21] Wang, J., and Guo, Y. Scrapy-based crawling and user-behaviour characteristics analysis on taobao. *International Conference on Cyber-Enabled Distributed Computing and Knowdedge Discovery* (2012), 44–52.

Appendix A

Online resources for this project

A.1 Sourceforge project

`http://sourceforge.net/projects/wwatch`

A.2 Program source code

The source code can be viewed online, via the URL given below, and it can be downloaded via the URL given in the section *Program download* (A.3).

`http://sourceforge.net/p/wwatch/code`

A.3 Program download

The download is provided as a Python package, which includes the source code of the program. The package also includes the source files for the program documentation, written as reStructuredText.[1]

`http://sourceforge.net/projects/wwatch/files`

[1] `http://docutils.sourceforge.net/rst.html`

A.4 Online program documentation

The program documentation can be viewed online, by following the URLs given below, and it can be downloaded as a PDF file, by following the URL given in in the section *Program download* (A.3).

A.4.1 User documentation

`http://wwatch.sourceforge.net/userdoc.html`

A.4.2 API

`http://wwatch.sourceforge.net/api.html`