UF Technical Report, Number TR98-007, Gainesville, FL, June 1998

# MOBILE WEB CRAWLING

Jan Fiedler and Joachim Hammer
University of Florida
Gainesville, FL 32611-6125
{jfiedler,jhammer}@cise.ufl.edu

## Abstract

Due to the enormous growth of the World Wide Web, search engines have become inevitable tools for Web navigation. In order to provide powerful search facilities, search engines maintain comprehensive indices of documents available on the Web. The creation and maintenance of these Web indices is done by Web crawlers, which recursively traverse and download portions of the Web on behalf of search engines.

Current Web crawlers operate in a centralized fashion, meaning that their entire functionality resides at the location of the search engine. Thus current crawlers are limited to rapid retrieval of Web pages and cannot perform any analysis of the data prior to downloading it. We argue that this could be a severe disadvantage since more information may be fetched than is eventually used by the index.

In this thesis we propose an alternative, more efficient approach based on mobile crawlers. We define mobility in the context of Web crawling as the ability of a crawler to transfer itself to the data source before the actual crawling process takes place. The major advantage of this approach is that the analysis portion of the crawling process can be done locally at the origin rather than remotely inside the Web search engine. Thus, network load can be reduced significantly by discarding irrelevant data prior to transmission which, in turn, can improve the performance of the crawling process. Our approach to Web crawling is particularly well suited for implementing crawlers that use “smart” crawling algorithms which require knowledge about the contents of documents.

In order to demonstrate the viability of our approach and to verify our hypothesis we have built a prototype architecture for mobile Web crawling that uses the UF intranet as its testbed. Based on this experimental prototype, we provide a detailed evaluation of the benefits of mobile Web crawling.

## 1. Introduction

### 1.1 Do we need Web Indices?

One of the characteristics of the World Wide Web is its decentralized structure. The Web basically establishes a very large distributed hypertext system, involving hundreds of thousands of individual sites. In fact, the number of individual sites as of April 1998 is estimated to be 1.6 million [BEL97]. Due to its distributed and decentralized structure, virtually anybody with access to the Web can add documents, links and even servers. Therefore, the Web is also very dynamic, changing 40% of its content within a month [KAH96].

Users navigate within this large information system by following hypertext links which connect different resources with one another. One of the shortcomings of this navigation approach is that it requires the user to traverse a possibly significant portion of the Web in order to find a particular resource (e.g., a document, which matches certain user criteria). Considering the growth rate of the Web (further discussed in section 1.3.1), it becomes harder and harder for an individual user to locate relevant information in a timely manner. Pinkerton [PIN94] refers to this problem as the resource discovery problem which, according to him, is comparable to the task of finding a book in a large library which does not maintain an index.

This comparison suggests an approach to solving the resource discovery problem using techniques which have proven to be effective in our daily life. By establishing a perfect comprehensive index of the Web which is accessible to all users, we could overlay the distributed structure of the Web with an additional centralized information system. This information system would solve the resource discovery problem by providing search functionality for the Web. This search functionality would free the user from browsing the Web in search of information, allowing him to locate relevant data using the centralized Web index.

Since maintaining indices for fast navigation within large collections of data has been proven to be an effective approach (e.g., library indices, book indices, telephone indices and database management systems), we see Web indices as an inevitable part of the Web. Therefore, we have to address the question of how such an index can be established in the highly distributed, decentralized and dynamic environment of the Web. Section 1.2 introduces the most commonly used technique. We then discuss some of the shortcomings of this technique in section 1.3. Keeping in mind the shortcomings of the current techniques and their implications with respect to the future growth of the Web, section 1.5 concludes this introduction with a motivation for a new, more efficient and therefore faster approach to Web crawling which is the main focus of this thesis.

## 1.2 Current Web Crawling Techniques

Web crawling can be characterized as a process which involves the systematic traversal of Web pages for the purpose of indexing the content of the Web. Consequently, Web crawlers are information discovery tools which implement certain Web crawling algorithms. To understand the general operation of a Web crawler let us look at the following generic structure of a crawling algorithm.

**Retrieval stage:** Since the ultimate goal of a crawler is to establish a Web index, the crawler has to retrieve the resources which will be part of the index. For example, in this stage a crawler might contact a remote HTTP (Hypertext Transfer Protocol) server, requesting a Web page specified by a URL (Uniform Resource Locator) address.

**Analysis stage:** After a certain resource has been retrieved, the crawler will analyze the resource in a certain way depending on the particular crawling algorithm. For example, in case the retrieved resource is a Web page, the crawler will probably extract hyperlinks and keywords contained in the page.

**Decision state:** Based on the results of the analysis stage, the crawler will make a local decision how to proceed in the crawling process. To continue our example from above, the crawler might identify some (or all) of the extracted hyperlinks as being candidates for the index by providing them as new input for the retrieval stage. This will basically restart the whole process again.

All crawlers examined in the context of this work follow this generic structure. The particular differences in crawling strategies are due to different implementations of the stages identified above. For example, breadth first and depth first crawlers usually only differ in their implementation of the decision stage by ordering extracted links differently. As another example consider fulltext and non-fulltext crawlers. Their implementation is likely to differ in the analysis stage, where non-fulltext crawlers extract certain page structures (e.g., page header and keywords) instead of keeping the source code of the page.

### *1.2.1. Generic Architecture*

Since the introduction of the World Wide Web Worm [MCB94] in 1994, one of the first Web search engines available, the primary research and development of search engines has been done in the commercial domain. Certainly, one reason for this is that search engine research is very expensive due to enormous resources needed to deal with the huge amount of information available on the Web. The effect of this is that only very little information about search engine and crawler technology is publicly available.

Only recently, large scale search engines and their crawlers received attention from a larger audience in the academic domain. This is due to the Google project [BRI97] conducted at Stanford University. The Google Web index currently contains 24 million pages and is therefore big enough to be a good representative of current search engines and crawlers as far as index size is concerned. For this reason we will use Google as our generic architecture to investigate the state of the art in Web crawling. Figure 1 depicts the part of the Google architecture which is relevant to Web crawling. Before we discuss the performance of this crawling architecture, let us briefly introduce some of the important components depicted in Figure 1.

**Crawler:** For high efficiency, Google uses multiple distributed crawlers for retrieving pages from the Web. This allows Google to distribute the load caused by crawling the Web among multiple hosts in a local network.

**Store Server:** The store server is responsible for storing pages retrieved by the crawlers into the local data repository. For storage efficiency, pages are compressed before being stored in the repository.

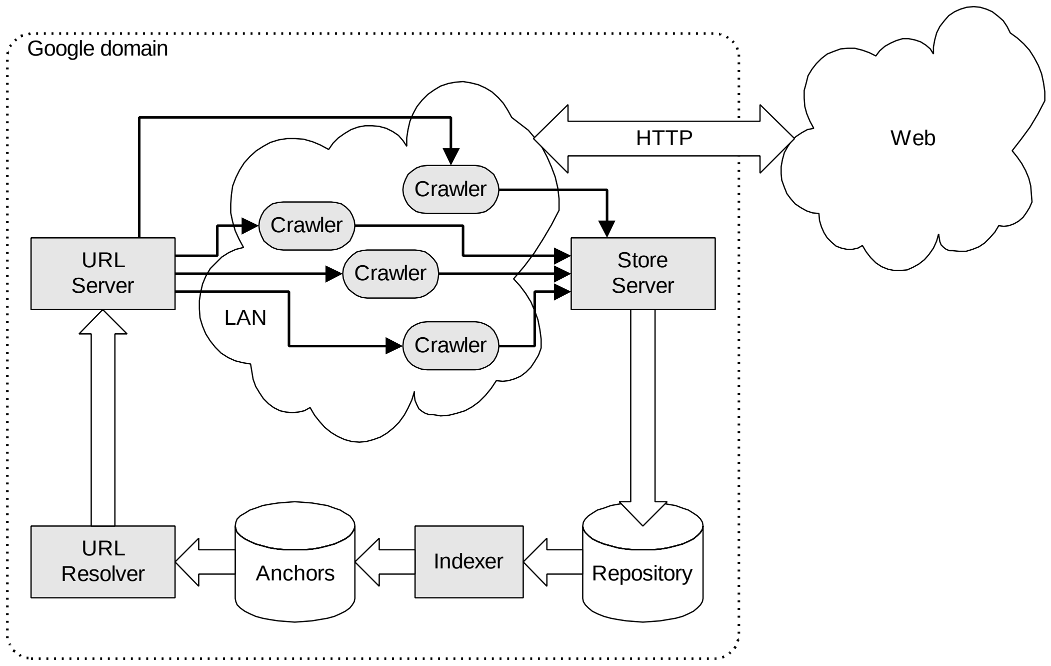

Figure 1: Google crawling architecture.

**Indexer:** The indexer performs a number of functions which are important to generate the Web index and are therefore not discussed here. However, there is one crucial function the indexer provides for the crawling process. This involves the parsing of the retrieved pages and the extraction of links. Identified links are stored in the anchor file for further processing by the URL resolver.

**URL Resolver:** The URL resolver reads the anchor file generated by the indexer and converts relative URLs into absolute URLs which then serve as new input to the crawling process.

**URL Server:** The URL server finally distributes the new links provided by the URL resolver to the crawlers running in the network.

Based on the fact that Google allows multiple crawlers to retrieve data in parallel, one might think that one can easily increase the crawling speed by increasing the number of crawlers. Unfortunately, this is not the case because several bottlenecks (e.g., local network speed, Web connection bandwidth, storage server speed) limit the throughput of the system. It is interesting to note that the authors of Google usually do not use more than 3 crawlers simultaneously. This is because each crawler manages about 300 Web connections at the same time. Combined, the three crawlers retrieve up to 100 Web pages (roughly 600Kbyte) per second. Thus, even without considering communication overhead, about 5Mbit of network bandwidth is consumed by the crawlers. If we add communication overhead, crawling traffic can easily saturate a 10Mbit Ethernet LAN (which is considered the standard network bandwidth for local networks) and thus significantly affect the Web performance experienced by other users.

The data presented above strongly suggests that Google's crawling system will experience scaling problems as soon as the size of the Google index (and thus the number of pages crawled) is increased to keep up with the growing Web. In this case the Web performance would have to improve significantly or the crawling process will need more time to achieve a reasonable coverage of the Web. The main problem here is that significant improvements in network performance are unlikely in the near term future and a larger time frame for crawling is unacceptable due to rapid index aging. We will discuss these problems further in section 1.3.1.

#### *1.2.2. Web Crawling Statistics*

Using the crawling throughput data of the Google architecture as discussed in section 1.2.1, we get a best case estimate of about 8.5 million retrieved pages per day for this architecture. By looking at the statistical data of big commercial search engines depicted in Table 1 [SUL98], we can validate our earlier assumption, that the Google crawling architecture is reasonably close to the approaches used by commercial systems. The implication of this similarity between Google and the commercial systems is that all of them are likely to face problems when attempting to keep pace with the growth of the Web.

Table 1: Crawling statistics for popular search engines.

| | *Altavista* | *Excite* | *HotBot* | InfoSeek | *Lycos* | *Northern Light* | Web Crawler |
|---|---|---|---|---|---|---|---|
| Index size (in million pages) | 10 | 55 | 110 | 30 | 30 | 30-50 | 2 |
| Daily crawl (in million pages) | 10 | 3 | 10 | - | 6-10 | - | - |
| Index age (in days) | 1-30 | 7-21 | 1-14 | 0-60 | 7-14 | 14 | 7 |

### 1.3 Drawbacks of current Crawling Techniques

As pointed out in the last sections, building a comprehensive fulltext index of the Web consumes significant resources of the underlying network. To keep the indices of a search engine up to date, crawlers are constantly busy retrieving Web pages as fast as possible. According to the figures presented in section 1.2.2, Web crawlers of big commercial search engines crawl up to 10 million pages per day. Assuming an average page size of 6K [BRI97] the crawling activities of a single commercial search engine adds a daily load of 60GB to the Web. This load is likely to increase significantly in the near future given the exponential growth rate of the Web. Unfortunately, improvements in network capacity did not keep pace with this development in the past and probably won't in the future. For this reason, we have considered it worthwhile to critically examine traditional crawling techniques in more detail in order to identify current and future weaknesses which we have addressed in our new, improved approach. In the following sections we discuss the main problems of the current Web crawling approach.

### *1.3.1. Scaling Issues*

Within the last couple of years, search engine technology had to scale dramatically to keep up with the growing amount of information available on the Web. One of the first Web search engines, the World Wide Web Worm [MCB94], was introduced in 1994 and used an index of 110,000 Web pages. Big commercial search engines in 1998 claim to index up to 110 million pages [SUL98]. This is an increase by factor 1000 in 4 years only. The Web is expected to grow further at an exponential speed, doubling its size (in terms of number of pages) in less than a year [KAH96]. By projecting this trend into the near future we expect that a comprehensive Web index will contain about 1 billion pages by the year 2000.

Another scaling problem is the transient character of the information stored on the Web. Kahle [KAH96] reports that the average online time of a page is as short as 75 days which leads to total data change rate of 600GB per month. Although changing pages do not affect the total index size, they cause a Web index to age rapidly. Therefore, search engines have to refresh a considerable part of their index frequently. Table 2 summarizes the current situation for search engines and provides estimates of what we should expect within the next couple of years. The material presented in Table 2 is based on current statistics provided by Sullivan [SUL98] and assumes a growing rate for the Web as discussed above.

Table 2: Web index size and update estimates.

| Year | Indexed Pages | Estimated Index Size | Daily page crawl | Daily crawl load |
|---|---|---|---|---|
| 1997 | 110 million | 700GB | 10 million | 60GB |
| 1998 | 220 million | 1.4TB | 20 million | 120GB |
| 1999 | 440 million | 2.8TB | 40 million | 240GB |
| 2000 | 880 million | 5.6TB | 80 million | 480GB |

There are two problems with scaling a search engine towards upcoming index sizes as presented in Table 2. First, the storage technology must scale such that it can provide access to several terabytes of data in a timely manner. Second, the crawling process has to improve significantly in order to establish Web indices of the projected size and, even more important, to keep them up to date. Using the estimates from Table 2, a Web crawler running in the year 2000 would have to retrieve Web data at a rate of 45Mbit per second in order to download the estimated 480GB per day. Looking at the fundamental limitations of storage technology and communication networks it is highly unlikely that Web indices as estimated in Table 2 can be implemented in the near future. Storage density increases about 50% per year, quadrupling in just over three years. Thus, the growing rate of storage technology does not keep pace with the projected growth of Web indices. Due to the even slower long term growing rate of communication networks, we do not expect that the Web can provide sufficient resources for the retrieval of such amounts of information in the near future.

### *1.3.2. Efficiency Issues*

Since Web crawlers generate a significant amount of Web traffic as pointed out in section 1.2.2, one might ask whether all the data downloaded by a crawler are really necessary. The answer to this question is almost always no.

In the case of specialized search engines (this are search engines which cover a certain subject area only) the answer is definitely no because these engines focus on a good coverage of specific subjects and are therefore not interested in all Web pages retrieved by their crawlers. It is interesting to note that the big commercial search engines which try to cover the whole Web are subject specific in some sense too. As an example for such a hidden subject specific criteria consider the language of a Web page. All general purpose search engines are only interested in Web pages written in certain languages (e.g., English, Spanish, German, French, but usually not Chinese). These engines are usually even less interested in pages which contain no textual information at all (e.g., frame sets, image maps, figures).

Unfortunately, current Web crawlers download all these irrelevant pages because traditional crawling techniques can not analyze the page content prior to page download. Thus, the data retrieved through current Web crawlers always contains some noise which consumes network resources without being useful. The noise level mainly depends on the type of search engine (general purpose versus specialized subject) and cannot be reduced using traditional crawling techniques.

### *1.3.3. Index Quality Issues*

In section 1.3.1 we outlined the limitations traditional search engines will face with respect to available bandwidth and growth of the Web. Even if the advances in storage and network capacity can keep pace with the growing amount of information available on the Web, it is questionable whether a larger index necessarily leads to better search results. Current commercial search engines maintain Web indices of up to 110 million pages [SUL98] and easily find several thousands of matches for an average query. From the user's point of view it doesn't make any difference whether the engine returned 10,000 or 50,000 matches because the huge number of matches is not manageable.

For this reason, we argue that the size of current Web indices is more than sufficient to provide a reasonable coverage of the Web. Instead of trying to accommodate an estimated index size of up to 1 billion pages for the year 2000, we suggest that more thought should be spent on how to improve the quality of Web indices in order to establish a base for improved search quality. Based on high quality indices (which preserve more information about the indexed pages than traditional indices do), search engines can support more sophisticated queries. As an example consider a Web index which preserves structural information (e.g., outgoing and incoming links) of the indexed pages. Based on such an index, it would be possible to narrow the search significantly by adding structural requirements to the query. For example, such a structural query could ask for the root page of a tree of pages such that several pages within the tree match the given keyword. Structural queries would allow the user to identify clusters of pages dealing with a certain topic. Such page clusters are more likely to cover the relevant material in reasonable depth than isolated pages.

High quality Web indices have considerably higher storage requirements than traditional indices containing the same number of pages because more information about the indexed pages needs to be preserved. For this reason, high quality indices seem to be an appropriate option for specialized search engines which focus on covering a certain subject area only. Using a high quality index a specialized search engine could provide much better search results (through more sophisticated queries) even though its index might contain significantly less pages than a comprehensive one.

Due to these advantages, we expect a new generation of specialized search engines to emerge in the near future. These specialized engines may challenge the dominance of today's big commercial engines by providing superior search results for specific subject areas. In section 1.3.2 we pointed out that even though the number of pages indexed by a specialized search engine is significantly smaller, the identification of relevant pages is still highly inefficient given today's crawling technology. This is because current crawlers are designed for building comprehensive Web indices. Traditional crawlers thus download all information regardless of its relevance with respect to a certain subject area. Therefore, we argue that a new crawling approach needs to be established which improves the efficiency of Web crawling especially for subject specific Web indices.

## 1.4 Future of Web Indices

Based on the problems of the current Web crawling approach we draw two possible scenarios for the future of search engines and their corresponding Web indices.

**Quantity oriented scenario:** The traditional search engines increase their crawling activities in order to catch up with the growing Web, requiring more and more information to be downloaded and analyzed by the crawlers. This adds an additional burden to the Web which already experiences an increase in traffic due to the increase in the number of users. It is questionable whether a quantity oriented search engine will be able to maintain a good coverage of the Web.

**Quality oriented scenario:** The traditional search engines realize that they are not able to cover the whole Web any more and focus on improving the index quality instead of the index coverage. Eventually, this transforms the general purpose search engines into specialized search engines which maintain superior coverage of a certain area of interest (e.g., economy, education or entertainment) rather than some coverage of everything. General purpose searches can still be addressed by meta search engines which use the indices of multiple specialized search engines.

Although we suspect a trend towards the second scenario, we consider it fairly uncertain which one will eventually take place. Therefore, we identify a new crawling approach which addresses both projected scenarios.

## 1.5 Introducing an alternative Crawling Approach

Given the problems of the current Web crawling techniques, we propose an alternative approach to Web crawling based on mobile crawlers. Crawler mobility allows for more sophisticated crawling algorithms which avoid the brute force strategy exercised by current systems. More importantly, mobile crawlers can exploit information about the pages being crawled in order to reduce the amount of data which needs to be transmitted to the search engine.

We discuss the general idea behind our approach and its potential advantages in 0. In 0 we introduce a prototype architecture which implements the proposed mobile crawling approach. In 0 we evaluate the advantages of our crawling approach by providing measurements from experiments based on our mobile crawler prototype system.

## 2. Related Research

### 2.1 Search Engine Technology

Due to the short history of search engines, there has been little time to research this technology area. One of the first papers in this area introduced the architecture of the World Wide Web Worm [MCB94] (one of the first search engines for the Web) and was published in 1994. Between 1994 and 1997, the first experimental search engines were followed by larger commercial engines such as WebCrawler, Lycos, Altavista, Infoseek, Excite and HotBot. As mentioned in section 1.2, there is very little information available about these search engines and their underlying technology. Only two papers about architectural aspects of WebCrawler [PIN94] and Lycos [MAU97] are publicly available on the Web. The Google project [BRI97] at Stanford University recently brought large scale search engine research back into the academic domain.

### 2.2 Web Crawling Research

Since crawlers are an inevitable part of a search engine, there is not much material available about commercial crawlers either. Again, a good information source is the Stanford Google project [BRI97] which we used as our primary architecture model in section 1.2.1. Based on the Google project, researchers at Stanford compared the performance of different crawling algorithms and the impact of URL ordering [CHO97] on the crawling process. A comprehensive Web directory providing information about crawlers developed for different research projects, can be found on the robots homepage [KOS97].

Another project which investigates Web crawling and Web indices in a broader context is the Harvest project [BOW95]. Harvest supports resource discovery through topic-specific content indexing made possible by a efficient distributed information gathering architecture. Harvest can therefore be seen as a base architecture upon which different resource discovery tools (e.g., search engines) can be built. A major goal of the Harvest project is the reduction of network and server load associated with the creation of Web indices. To address this issue, Harvest uses distributed crawlers (called gatherers) which can be installed at the site of the information provider to create and maintain an provider specific index. The indices of different providers are then made available to external resource discovery systems by so called brokers which can use multiple gatherers (or even other brokers) as their information base.

Beside technical aspects there is a social aspect to Web crawling too. As pointed out in section 1.2.2, a Web crawler consumes significant network resources by accessing Web documents at a fast pace. More importantly, by downloading the complete content of a Web server, a crawler might significantly hurt the performance of the server. For this reason, Web crawlers have earned a bad reputation and their usefulness is sometimes questioned as discussed by Koster [KOS95]. To address this problem, a set of guidelines for crawler developers has been published [KOS93]. In addition to these general guidelines a specific Web crawling protocol, the Robot Exclusion Protocol [KOS96], has been proposed by the same author. This protocol enables webmasters to specify to crawlers which pages not to crawl. However, this protocol is not yet enforced and Web crawlers implement it on a voluntary basis only.

## 2.3 Rule Based Systems

Crawler behavior can be expressed through rules that tell a crawler what it should do versus developing a sequential algorithm which gives an explicit implementation for the desired crawler behavior. We will discuss this issue in more detail in section 4.2 when introducing our approach to crawler specification.

An example for a rule based system is CLIPS [GIA97] (C Language Integrated Production System) which is a popular expert system developed by the Software Technology Branch at the NASA/Lyndon B. Johnson Space Center. CLIPS allows us to develop software which models human knowledge and expertise by specifying rules and facts. Therefore, programs do not need a static control structure because they are specified as a set of rules which reason about facts and react appropriately.

In the context of our prototype system we use a Java version of CLIPS called Jess (Java Expert System Shell) [FRI97]. Jess provides the core CLIPS functionality and is implemented at the Sandia National Laboratories. The main advantage of Jess is that it can be used on any platform which provides a Java virtual machine which is ideal for our purposes.

## 2.4 Mobile Code

Mobile code has become fairly popular in the last couple of years especially due to the development of Java [GOS96]. The best example are Java applets which are small pieces of code, downloadable from a Web server for execution on a client. The form of mobility introduced by Java applets is usually called remote execution, since the mobile code gets executed completely once it has been downloaded. Since Java applets do not return to the server, there is no need to preserve the state of an applet during the transfer. Thus, remote execution is characterized by stateless code transmission.

Another form of mobile code, called code migration, is due to mobile agent research. With code migration it is possible to transfer the dynamic execution state along with the program code to a different location. This allows mobile agents to change their location dynamically without affecting the progress of the execution. Initial work in this area has been done by General Magic [WHI96]. Software agents are an active research area with lots of publications focusing on different aspects of agents such as agent communication, code interoperability and agent system architecture. Some general information about software agents can be found in papers from Harrison [HAR96], Nwana [NWA96] and Wooldridge [WOO95]. Different aspects and categories of software agents are discussed by Maes ([MAE94] and [MAE95]). Communication aspects of mobile agents are the main focus of a paper by Finin [FIN94].

# 3. A Mobile Approach to Web Crawling

## 3.1 Mobile Crawling Overview

In this chapter we introduce our Web crawling approach which uses mobile crawlers to address the problems of the traditional crawling techniques identified in section 1.3.

By looking at the general anatomy of traditional search engines as introduced in section 1.2.1, we realize that their architecture is strictly centralized while the data being accessed by search engine crawlers is highly distributed. We argue that this centralized architecture is a mismatch for the distributed organization of the Web because it requires data to be downloaded before it can be processed. In particular, all Web pages, whether or not it is relevant for the search engine, have to be downloaded **before** they can be analyzed and stored. This is not a big problem if the ultimate goal of the search engine is to cover the whole Web. In this case every page has to be downloaded and stored by a crawler anyway.

The shortcomings of the current approach become clear when we look at specialized search engines which are only interested in certain Web pages. If we use a stationary crawler, we would download a lot of pages which are discarded immediately because they do not meet the subject area of the search engine. This is because a stationary crawler has to download a page before a decision can be made whether or not the page is relevant. Obviously, this behavior is not very desirable because large amounts of bandwidth might be wasted by downloading irrelevant information.

The Web crawling approach proposed in this work addresses this problem by introducing mobile crawlers. This breaks the centralized architecture of traditional search engines by making the data retrieval component, the Web crawler, distributed. We define mobility in the context of Web crawling as the ability of a crawler to transfer itself to the data source (e.g., a Web server) before the actual crawling process is started on that Web server. Thus, mobile crawlers are able to move to the resource which needs to be accessed in order to take advantage of local data access. After accessing a resource, mobile crawlers move on to the next server or to their home system, carrying the crawling result in the memory. Figure 2 clarifies the role of mobile crawlers and depicts the decentralized data retrieval architecture as established by mobile crawlers.

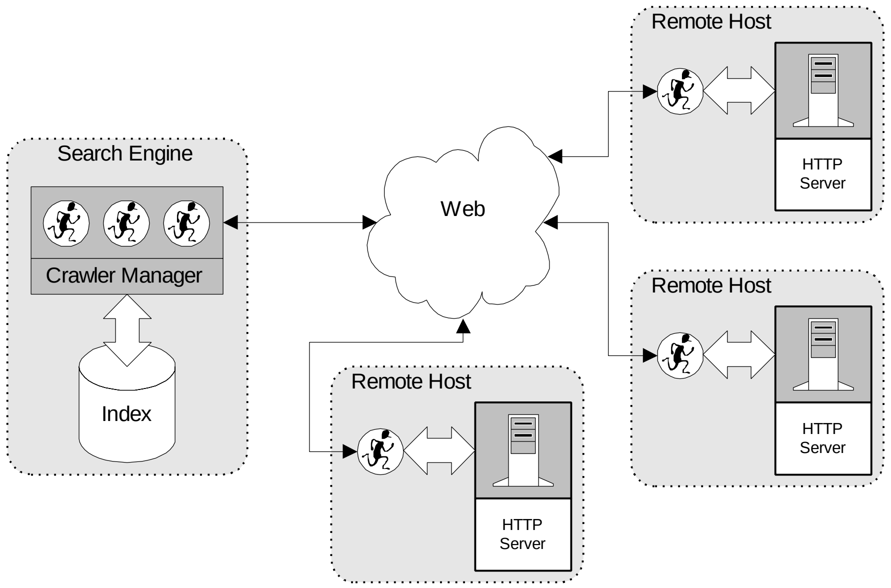

Figure 2: Mobility based crawling approach.

The main advantage of the approach depicted in Figure 2 is that it allows us to distribute crawling functionality within a distributed system such as the Web. We discuss the specific advantages of this approach in detail in section 3.2.

## 3.2 Mobile Crawling Advantages

### *3.2.1. Localized Data Access*

The main task of stationary crawlers in traditional search engines is the retrieval of Web pages on behalf of the search engine. The communication protocol to be used for the actual data retrieval is HTTP (Hypertext Transfer Protocol) [BER96]. HTTP is an application level protocol specifically designed for distributed, collaborative, hypertext information systems such as the World Wide Web. The HTTP protocol is based on a request/response paradigm with certain formats for request and response messages. A HTTP client retrieves a Web page by establishing a connection with the server and requesting the document by sending a request message. Then the HTTP client waits for the server's response message containing the requested data.

In the context of traditional search engines the HTTP client is a stationary crawler which tries to recursively download all documents managed by one or more Web servers. Due to the HTTP request/response paradigm, downloading the contents from a Web server involves significant overhead due to request messages which have to be sent for each Web page separately. Figure 3 depicts this request based scheme imposed by HTTP in the context of Web crawling. Figure 3 shows the overhead involved in the retrieval of Web pages in terms of messages send over the network. Since HTTP request messages can be several hundred bytes in size (depending on the length of the URL address of the requested document), a significant portion of the available bandwidth between the crawler and the HTTP server is consumed by those messages.

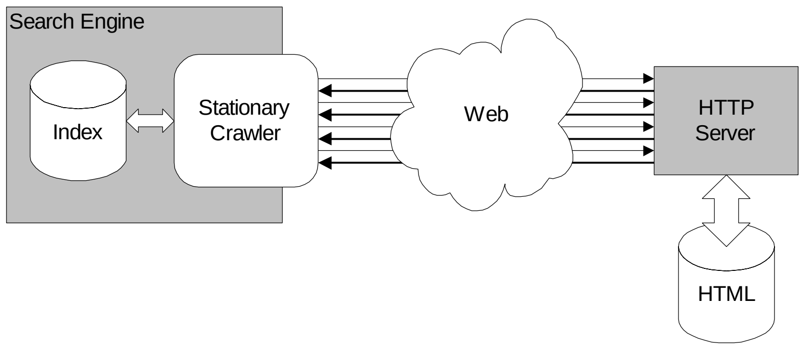

Figure 3: HTTP based data retrieval using stationary crawlers.

The situation is actually worse than depicted in Figure 3 because HTTP is an application level protocol. This means that all HTTP request messages are encoded into the message format of the underlying transport protocol (usually TCP) introducing additional overhead due to protocol specific header and framing information.

Using a mobile crawler we reduce the HTTP overhead by transferring the crawler to the source of the data. The crawler can then issue all HTTP requests locally with respect to the HTTP server. This approach still requires one HTTP request per document but there is no need to transmit these requests over the network anymore. Figure 4 summarizes the data retrieval process based on mobile crawlers as introduced above. The main advantage of the approach depicted in Figure 4 is that it makes use of local rather than remote data access. Instead of issuing requests over the network, a mobile crawler first migrates to the data source and accesses the information locally. A mobile crawler thus saves bandwidth by eliminating Web traffic caused by HTTP requests.

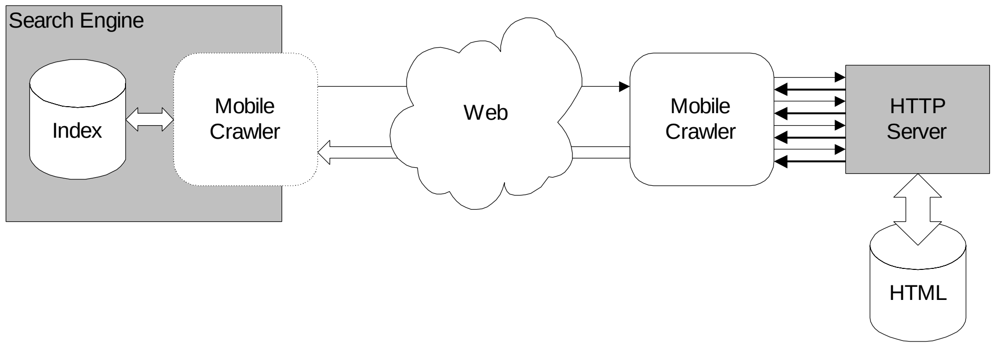

Figure 4: HTTP based data retrieval using mobile crawlers.

Naturally, this approach only pays off if the reduction in Web traffic due to local data access is more significant than the traffic caused by the initial crawler transmission.

### *3.2.2. Remote Page Selection*

By using mobile crawlers we can distribute the crawling logic (i.e. the crawling algorithm) within a system of distributed data sources such as the Web. This allows us to transform Web crawlers from simple data retrieval tools into more intelligent components which can exploit information about the data they are supposed to retrieve. Crawler mobility allows us to move the decision whether or not certain pages are relevant to the data source itself.

The most important application of this feature is remote page selection. Once a mobile crawler has been transferred to a Web server, it can analyze each Web page **before** doing anything with it which would require network resources. By analyzing a Web page, the crawler can determine whether the page is relevant with respect to the search engine the crawler is working for. Web pages considered to be relevant are stored within the crawler and eventually transmitted over the network when the mobile crawler returns to its origin location.

By looking at remote page selection from a more abstract point of view, it compares favorably with classical approaches in database systems. If we consider the Web as a large remote database, the task of a crawler is akin to querying this database in order to extract certain portions of it. In this context, the main difference between traditional and mobile crawlers is the way queries are issued. Traditional crawlers implement the data shipping approach of database systems because they download the whole database before they can issue a queries to identify the relevant portion. If the query is very specific (e.g., establish an index of all English health care pages), a major part of the database has been downloaded without being useful in any way. In contrast to this, mobile crawlers use the query shipping approach of database systems because all the information needed to identify the relevant data portion is transferred directly to the data source along with the mobile crawler. After the query has been executed remotely, only the query result is transferred over the network and can be used to establish the desired index without requiring any further analysis.

Since the query shipping approach has proven to be more efficient in the context of database systems (e.g., SQL servers), we consider its application in the context of Web crawling to be superior with respect to the traditional approach. Section 5.2 provides some statistics on how remote page selection helps to reduce Web traffic while still covering all relevant data.

### *3.2.3. Remote Page Filtering*

Remote page filtering extends the concept of remote page selection introduced in section 3.2.2 to the contents of a Web page. The idea behind remote page filtering is to allow the crawler to control the granularity of the data it retrieves.

With stationary crawlers, the granularity of retrieved data is the Web page itself. This is because HTTP allows page-level access only. For this reason, stationary crawlers always have to retrieve a whole page before they can extract the relevant page portion. Depending on the ratio of relevant to irrelevant information, significant portions of network bandwidth are wasted by transmitting useless data.

A mobile crawler addresses this problem through its ability to operate directly at the data source. Once a mobile crawler has been transferred to a remote HTTP server, it can retrieve Web pages by issuing local HTTP requests. The retrieval of the complete page by a mobile crawler does not require any network resources beside the cost of the initial crawler transmission. After retrieving a page, a mobile crawler can filter out all irrelevant page portions keeping only information which is relevant with respect to the search engine the crawler is working for.

An example of potential savings due to remote page filtering is a search engine index which relies on the page URL, page title and a set of page keywords only. Instead of transmitting the whole page (average size 6000 byte [BRI97]) a mobile crawler would discard all irrelevant information and keep only the information which is needed to establish the search engine index. Assuming a page URL of 60 characters, page title length of 80 characters and a set of 15 keywords with an average size of 10 characters each, the mobile crawler needs to keep only 290 bytes of the total page.

Thus, remote page filtering is especially useful for search engines which use a specialized representation for Web pages (e.g., URL, title, modification date, keywords) instead of storing the complete page source code. Due to the huge storage requirements of fulltext Web indices and the rapid growing rate of the Web, we expect a paradigm shift towards such non-fulltext indices for the future.

### *3.2.4. Remote Page Compression*

With remote page selection and filtering as introduced in section 3.2.2 and 3.2.3 we introduced two techniques aimed towards the reduction of network traffic caused by Web crawlers. The techniques discussed so far perform well in the context of specialized search engines which cover a certain portion of the Web only. In these cases a mobile crawler can use remote page selection and filtering to identify the relevant portion of the data right at the data source before carrying it to the search engine. The situation is very different for a crawler which is supposed to establish a comprehensive fulltext index of the Web. In this case, as many Web pages as possible need to be retrieved by the crawler. Techniques like remote page selection and filtering are not applicable in such cases since every page is considered to be relevant.

In order to reduce the amount of data that has to be transmitted back to the crawler controller, we introduce remote page compression as a basic feature of mobile crawlers. Once a mobile crawler finished crawling a HTTP server, it has identified a set of relevant Web pages which is kept in the crawler's data repository. In order to reduce the bandwidth required to transfer the crawler along with the data it contains back to the search engine, the mobile crawler applies compression techniques to reduce its size prior to transmission. Note that this compression step can be applied independently form remote page selection and filtering featured by mobile crawlers. Thus, remote page compression reduces Web traffic for mobile fulltext crawlers as well as for mobile subject specific crawlers.

Remote page compression makes mobile crawling an attractive approach even for traditional search engines which do not benefit from remote page selection and filtering due to their comprehensive fulltext indexing scheme. Due to the fact that Web pages are human readable ASCII messages, we expect excellent compression ratios with standard compression techniques. We evaluate the reduction of Web traffic due to remote page compression in section 5.4.

## 3.3 Mobile Crawling Example

In order to demonstrate the advantages of mobile crawling, we present the following example. Consider a special purpose search engine which tries to provide high quality searches in the area of health care. The ultimate goal of this search engine is to create an index of the part of the Web which is relevant to health care issues. The establishment of such a specialized index using the traditional crawling approach is highly inefficient. This inefficiency is because traditional crawlers would have to download the whole Web page by page in order to be able to decide whether a page contains health care specific information. Thus, the majority of downloaded pages would not be stored in the index and would be discarded.

In contrast, a mobile crawler allows the search engine programmer to send a representative of the search engine (the mobile crawler) to the data source in order to filter it for relevant material before transmitting it back to the search engine. In the context of our example, the programmer would instruct the crawler to migrate to a Web server in order to execute the crawling algorithm right at the data source. An informal description of the remotely executed crawling algorithm could look like the following pseudocode.

```
/**
* Pseudocode for a simple subject specific
* mobile crawler.
*/

migrate to web server;
put server url in url_list;

for all url ∈ url_list do begin

  // *** local data access
  load page;

  // *** page analysis
  extract page keywords;
  store page in page_list if relevant;

  // *** recursive crawling
  extract page links;
  for all link ∈ page do begin
    if link is local then
      add link to url_list;
    else
      add link to external_url_list;
  end
end
```

Please note that this is very similar to an algorithm executed by a traditional crawler. The important difference is that our crawler gets executed right at the data source by the mobile crawler. The crawler analyzes the retrieved pages by extracting keywords. The decision, whether a certain page contains relevant health care information can be made by comparing the keywords found on the page with a set of predefined health care specific keyword known to the crawler. Based on this decision, the mobile crawler only keeps pages which are relevant with respect to the subject area. As soon as the crawler finishes crawling the whole server, there will be a possibly empty set of pages in its memory. Please note that the crawler is not restricted to having Web pages in its memory only. Any data which might be important in the context of the search engine (e.g., page metadata, Web server link structure) can be represented in the crawler memory. In all cases, the mobile crawler is compression to significantly reduce the data to be transmitted. After compression, the mobile crawler returns to the search engine and is decompressed. All pages retrieved by the crawler are then stored in the Web index. Please note, that there are no irrelevant pages since they have been discarded before transmission by the mobile crawler. The crawler can also report links which were external with respect to the Web server crawled. The host part of these external addresses can be used as migration destination for future crawls by other mobile crawlers.

By looking at the example discussed above, the reader might get an idea about the potential savings of this approach. In case a mobile crawler does not find any useful information on a particular server, nothing beside the crawler code would be transmitted over the network. If every single page of a Web server is relevant, a significant part of the network resources can be saved by compressing the pages prior to transmission. In both of these extreme cases, the traditional approach will produce much higher network loads. We will provide an analysis of the benefits of mobile crawling in 0.

# 4. An Architecture for Mobile Web Crawling

## 4.1 Architecture Overview

In 0 we defined the goal of our system as being a framework for supporting the discovery and the effective retrieval of information stored on the Web. We proposed the use of mobile crawlers as information retrieval tools for such a framework. An architecture implementing the proposed framework has to provide an environment which supports the execution of crawlers at remote locations. In addition to this execution environment we have to provide an application oriented framework which supports the creation and management of mobile crawlers. In our system architecture, we strictly distinguish between the distributed crawler runtime environment and the application framework architecture which uses the runtime environment to achieve application specific goals. Figure 5 depicts the overall system architecture and visualizes the architectural separation mentioned above. The conceptual separation into two distinct but cooperating architectures is carried through in the structure of the following sections. After discussing the purpose of the crawler runtime environment in section 4.1.1, we introduce the main ideas of the application oriented framework in section 4.1.2. Sections 4.3 through 4.7 focus on specific design issues of the most essential components used in our system.

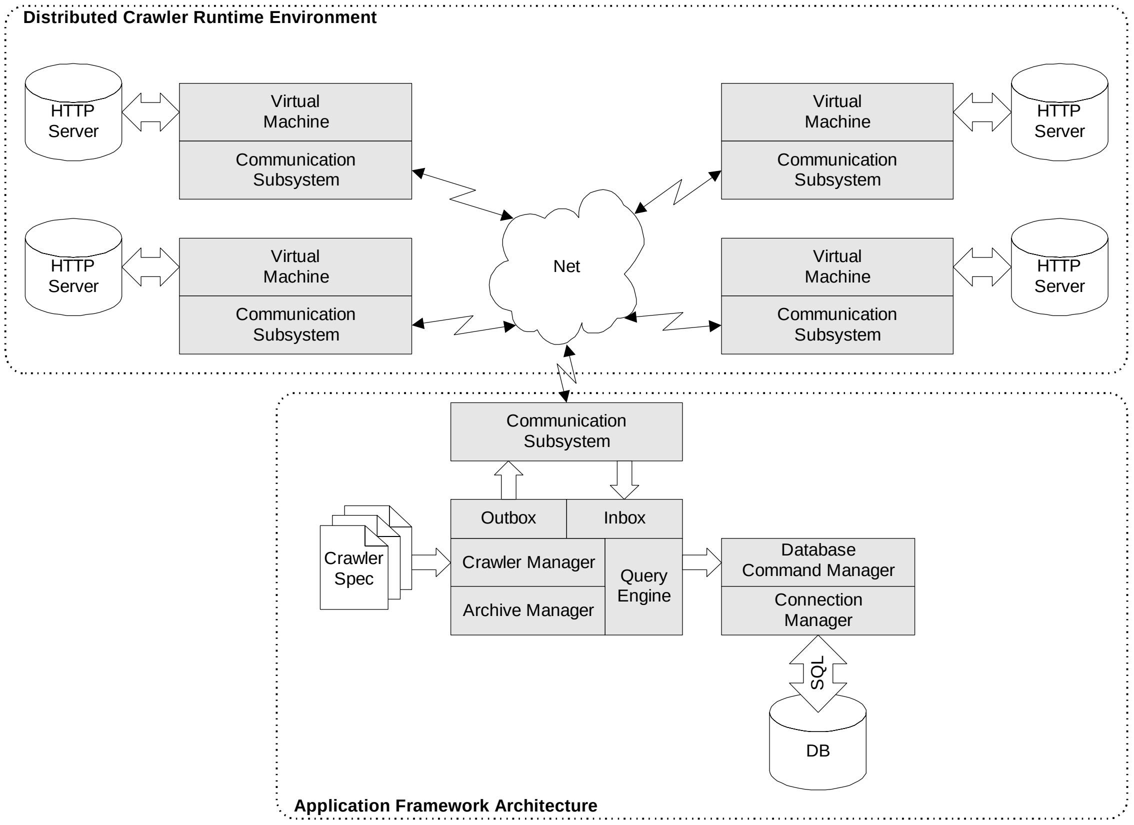

Figure 5: System Architecture Overview

### *4.1.1. **Distributed Crawler Runtime** Environment Overview*

The use of mobile crawlers for information retrieval requires an architecture which allows us to execute code (i.e. crawlers) on remote systems. Since we concern ourselves mainly with information to be retrieved from the Web, the remote systems of interest are basically Web servers. To feature remote crawling as outlined in 0, we would need to modify Web servers such that they could execute pieces of code sent to them over the network. Since modifications to the diverse base of installed Web servers was not a feasible solution for our project, we decided to provide a simple runtime environment for remote crawler execution. Our crawler runtime environment can be installed easily on any host wishing to participate in our experimental testbed. A host running our runtime environment becomes a part of our mobile crawling system and is able to retrieve and execute mobile code. The only purpose of the runtime environment is crawler execution. It does not insert new mobile crawlers into the Web (i.e. into the mobile crawler enabled subweb) nor does it initiate any arbitrary crawler transfers. The creation and management of mobile crawlers is considered to be an application specific issue to be addressed by the user application.

The mobile code architecture consists of two main components. The first component is the communication subsystem which primarily deals with communication issues and provides an abstract transport service for all system components. The second and more important component is the virtual machine which is responsible for the execution of crawlers. We refer to the combination of both components as the runtime environment for mobile crawlers since both are needed to provides the functionality necessary for the remote execution of mobile code. Both components are discussed in more detail in sections 4.3 and 4.4 respectively.

#### *4.1.2. Application Framework Architecture Overview*

The ultimate goal of our application framework architecture is to support the detection and retrieval of context specific information on the Web. Since we want to use the mobile crawling approach as introduced in section 3.1, we need to install crawler runtime environments on or close to (in terms of bandwidth) Web servers to establish a mobile code architecture as described in the previous section. From the application framework point of view, the mobile code architecture establishes a distributed execution environment in which application specific crawlers can operate. Thus, the application framework architecture must be able to create and manage mobile crawlers and has to control their migration to appropriate locations.

Since we want to be able to analyze the data we retrieve through our mobile crawlers, the application framework architecture also serves as a database frontend which is responsible for the appropriate transformation of retrieved data into a database scheme. The design of a suitable database scheme depends heavily on the kind of analysis to be performed on the data later on. Although this is not the primary focus of this work, we address this issue in order to be able to evolve the database scheme in case new analysis need to be performed featuring new kinds of data not yet accommodated by the database. All these issues are addressed by the archive manager architecture which is discussed in detail in section 4.7.

## 4.2 Crawler Specification

In order to create and execute crawlers within our system, we have to specify what a crawler is supposed to do by providing a crawler specification. One can think of the crawler specification as the internal program of a crawler. Crawler specifications play an important role in our system since they determine the crawler behavior. For this reason, we have to find an easy but powerful mechanism to describe crawler behavior.

### *4.2.1. Specification Language*

If we look closer at the common tasks and constraints a crawler has to implement, we realize that a crawler is a highly reactive piece of software which responds to certain events within its environment by performing appropriate operations.

This fact makes it inconvenient to specify crawler behavior as a sequential program using imperative programming languages. Such specifications would be dominated by deeply nested loops and if-then-else constructs which try to capture all the constraints a crawler is supposed to adhere to. There is another important issue which makes this approach unattractive. A crawler which relies on the traditional programming paradigm has a fairly complex runtime state (i.e. local variables, program counter, stack and heap), making it nearly impossible to transfer the crawler state to another machine. Since crawler mobility is the main feature of our crawling approach, this would make the implementation of our system unnecessarily complex. For this reason, we looked for an alternative approach to crawler specification.

By analyzing common crawling algorithms we realized, that crawler behavior can be easily expressed by formulating a set of simple rules which express how the crawler should respond to certain events or situations. For example, it is sufficient to provide a rule which states that a page should be loaded whenever a new URL address has been found. This simple rule replaces a complete loop in the traditional approach. Another simple rule could state that new URL addresses can be found by scanning pages which have been retrieved before. Both rules together already establish an easy crawler specification. Due to the lack of an explicit control structure, rule based specifications are easy to write and understand.

Due to the ease of generating specifications based on simple rules, we decided to use a rule based specification approach as implemented by artificial intelligence and expert systems. In particular, we decided to use the rule based language of CLIPS [GIA97] to specify crawler behavior. An important advantage of this approach is that it supports crawler mobility very well. Since a rule based crawler specification basically consist of a set of independent rules without explicit control structure, there is no real runtime state. By specifying crawlers using a rule based language we get crawler mobility almost for free.

### *4.2.2. Data Representation*

Due to the lack of local variables and control structure in a rule based program, we need to find another way to represent data relevant for the crawling algorithm. Virtually all rule based systems require the programmer to represent data as facts. Facts are very similar to data structures in traditional programming languages and can be arbitrarily complex. Rules, which will be discussed further in section

4.2.3, depend on the existence of facts as the data base for reasoning. For example, a rule (e.g., load-page-rule) can be written such that it gets activated as soon as a particular fact (e.g., new-URL-address-found) is available.

Rule based systems distinguish between ordered and unordered facts. Unordered facts are very similar to data structures in traditional programming language because the structure of an unordered fact has to be declared before usage. Figure 6 shows an example of an unordered fact declaration in CLIPS.

```
// *** DECLARATION
(deftemplate PAGE
  (slot status-code    (type INT))
  (slot location       (type STRING))
  (slot last-modified  (type INT))
  (slot content-type   (type STRING))
  (slot content-lenght (type INT))
  (slot content        (type STRING))
)


// *** RUNTIME REPRESENTATION
(PAGE (status-code 200)(location "http://www.cise.ufl.edu")(last-modified 7464542245)
      (content-type "text/html")(content-length 4123)(content "<HTML>...</HTML>"))
```

Figure 6: Unordered facts in CLIPS.

As we can see in the code fragment above, unordered facts contain named fields (called slots in CLIPS) which have certain types. Since every field can be identified by its name, we can access each field separately.

Ordered facts do not require a declaration prior to usage. This implies that ordered facts do not have fields names and field types. The implication of this is that fields of an ordered fact can not be accessed independently. Field access must be done based on the order of fields. It is the responsibility of the programmer to use the fields of ordered facts in a consistent manner. For example, an ordered version of the unordered fact given above would look like the following.

```
// *** DECLARATION
(no declaration required)

// *** RUNTIME REPRESENTATION
(PAGE 200 http://www.cise.ufl.edu 7464542245 "text/html" 4123 "<HTML>...</HTML>")
```

Figure 7: Ordered facts in CLIPS.

As shown in Figure 7, we need to know the position of a field within the fact before we can access it. Since this is inconvenient for structured data, ordered facts should be used to represent unstructured data only. Good examples for unstructured data are events and states which can be easily represented by ordered facts. Ordered facts are especially useful for controlling the activation of rules whereas unordered facts represent data upon which rules will operate.

### *4.2.3. Behavior Specification*

This section focuses on the second and more important part of a crawler specification, crawler rules. Rules in a rule based system establish something very similar to IF-THEN statements in traditional programming languages. Once a rule has been specified, the rule based system constantly checks whether the "IF-part" of the rule has become true. If this is the case, the "THEN-part" gets executed. To put it in a more formal way, we state that each rule has two parts, a list of patterns and a list of actions. Before the action list of a rule gets executed, the list of patterns has to match with facts such that each pattern evaluates to true. The action part of a rule might create new facts which might allow other rules to execute. Therefore, the intelligence of a rule based program is encoded in the program's rules. Also, by creating new facts in order to allow other rules to execute, we exercise a kind of implicit flow control. Figure 8 shows an example rule in CLIPS notation.

```
(defrule LoadPageRule
  (todo ?url)
  (not (done ?url))
  =>
  ;load the page
  (assert (done ?url))
)
```

Figure 8: Rules in CLIPS.

The example rule in Figure 8 gets executed as soon as facts can be found which match with the two patterns ("todo" and "not done"). The meaning of the ordered fact "todo" in this context is that a certain page needs to be retrieved from the Web. Before retrieving a page the rule makes sure that the particular page was not downloaded before. This is indicated by the absence of the ordered fact "done". For both patterns, "?url" is a variable which carries the value of the first field of the ordered fact which happens to be the URL address of the page. If both patterns can be matched with facts from the fact base, the rule action part gets executed. The action part will load the page (code omitted) and create a new "done" fact, indicating that the page has been retrieved.

## 4.3 Communication Subsystem

The communication subsystem implements a transparent and reliable communication service for the distributed crawler runtime environment as well as the application framework architecture. The communication service provided by the communication subsystem has to work on a high level of abstraction because we need to shield the remaining system components from low level communication issues such as protocol restrictions (e.g., packet size restrictions) and protocol specific addressing modes (e.g., Uniform Resource Locators versus TCP port numbers). The communication layer is built on top of common lower level communication services such as TCP, UDP, and HTTP and provides a transparent abstraction of these services within our system. The approach of using several lower level communication services to establish a high level communication service has the following advantages:

**Flexibility:** Using different communication protocols allows us to exploit special properties of the different protocols. Thus, we can always select the communication protocol which best suits the needs of the transmitting component or the data to be transmitted. The decision which protocol to use for a particular transfer is based on specific criteria. Considering data size as an example for such a criteria, the communication layer chooses an appropriate communication protocol according to the size of the data to be sent. The communication layer would choose the fast UDP protocol in case of small data portions (i.e. data portions smaller than the maximal UDP packet size) and switch to the slower but more reliable TCP protocol in case the data size exceeds a certain threshold (and thus needs to be transmitted as a data stream). The decision on which protocol to use can be based on different criteria such as security constraints, data transfer reliability and network architecture constraints.

**Reliability:** The use of several communication protocols which can be exchanged dynamically improves the reliability of our communication architecture in case of failures of a particular protocol. Facing a communication failure when using a certain protocol, the communication layer can recover by switching to another protocol dynamically. Thus, communication facilities remain available for all other components within our system.

**Extensibility:** The use of multiple communication protocols allows a system to evolve over time by using a different set of protocols as new or improved protocols become available. Thus, a system can address changes in the underlying network architecture by simply switching to a protocol which can deal with the changes made. As an example for such a change in the network architecture consider the installation of a firewall between the Internet and a corporate network. A firewall usually does not allow arbitrary TCP traffic which cripples applications relying exclusively on TCP communication. Applications using a dynamically extensible set of communication protocols can adjust to the new situation by transmitting their data using a different protocol (e.g., HTTP which can usually pass firewalls).

### *4.3.1. Communication Layer Architecture*

Due to the advantages of the multiple protocol approach outlined above, we decided to incorporate these ideas in the design of our communication subsystem. The communication system of our application is designed as a layer upon which all other system components operate. Specifically, the communication layer implements a protocol independent, abstract communication service which can be used by all other system components for data transfer. Figure 9 depicts this 3-tier architecture together with a set of components which each implement a certain communication protocol.

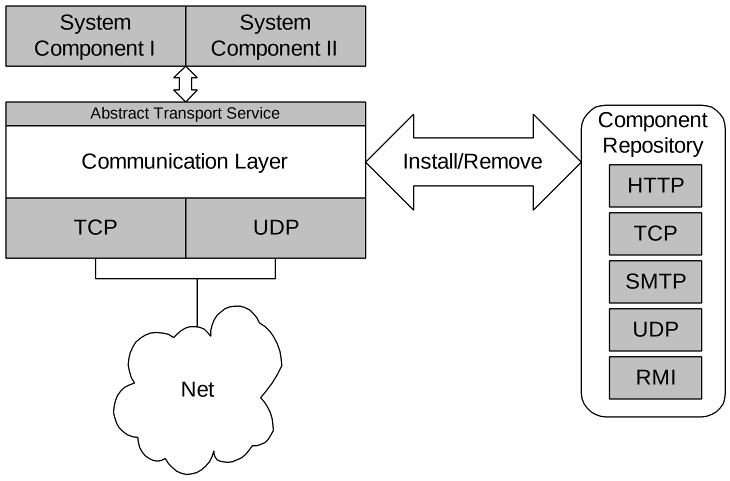

Figure 9: Communication layer architecture.

Please note that the communication layer does not implement any real communication protocols and therefore can not communicate on its own. Actual communication is established through dedicated communication components which can be added to the communication layer dynamically. Each communication component implements a certain protocol extending the set of protocols available to the communication layer. The set of communication components actually used by the communication layer depends on the kind of data transmitted, the network architecture the system is operating on, and administrative issues. The process of selecting an appropriate protocol is further discussed in section 4.3.3.

### *4.3.2. Communication Layer Addressing*

Section 4.3.1 introduced communication components which can be added to the communication layer dynamically to extend the communication capabilities of our system. This approach raises the question of how to address a certain communication layer. Since the set of communication protocols used by the communication layer changes dynamically and new protocols can be added to the system as needed the address mechanism of the communication layer must be able to accommodate these changes dynamically. This becomes clear as soon as we consider the significant differences between addressing information used by different protocols. As a simple example compare TCP and HTTP address information. TCP needs the name and the port number of the remote host whereas HTTP uses the URL (Uniform Resource Locator) notation to specify an abstract remote object. The address structure for our communication layer has to accommodate both because both might be installed as protocols to be used by the communication layer.

To allow a dynamic address structure we use abstract address objects. The only static information defined in our abstract address objects is a name attribute (e.g., "TCP" for a TCP address encapsulated in the object) which is used to distinguish different address types. The remaining protocol specific address information such as hostname and port versus URL is encoded dynamically as attribute/value pairs. Thus, each communication component can create its own address by using its protocol specific attributes. For example, all TCP communication components use the attribute "HOSTNAME" for the name of the host their TCP socket was created on and the attribute "PORT" for the associated TCP port number. Using this approach the address of the communication layer can be defined as the union of all the address objects of communication components used by the communication layer. Figure 10 depicts an example of such an address.

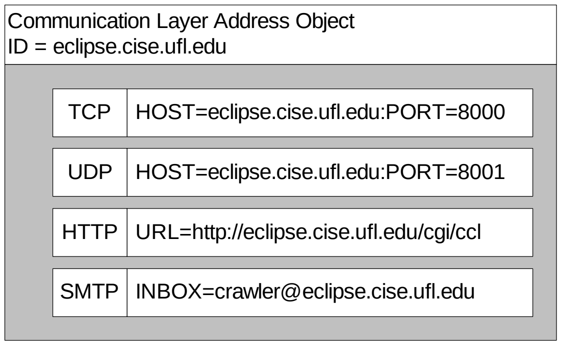
Communication Layer Address Object
ID = eclipse.cise.ufl.edu
TCP
HOST=eclipse.cise.ufl.edu:PORT=8000
UDP
HOST=eclipse.cise.ufl.edu:PORT=8001
HTTP
URL=http://eclipse.cise.ufl.edu/cgi/ccl
SMTP
INBOX=crawler@eclipse.cise.ufl.edu

Figure 10: Abstract communication layer address object.

The address object in Figure 10 contains four different protocol addresses with their protocol specific address information. Using this address structure, we accommodate an arbitrary number of protocols with their protocol specific addressing attributes. The example address given in Figure 10 suggests that it belongs to a communication layer installed at host eclipse in the computer science department at the University of Florida. The example address also indicates that the communication layer it belongs to has four communication components installed which allow data transfers using TCP, UDP, HTTP and SMTP respectively.

### *4.3.3. Communication Layer Protocol*

So far, we have focused on the individual layers in the communication subsystem architecture. Within this section we shift our focus to the dynamic processes involved in the communication among several distributed communication layers. This section explains, how actual data transfer is performed using the architecture introduced in section 4.3.1.

Since the set of protocols used by the communication layer can change dynamically (due to new installed or removed communication components) a communication layer trying to initiate a data transfer can not make any assumptions about protocols supported by the intended receiver of the data (which is another communication layer). Thus, the protocol to be used for a particular data transfer can not be determined by looking at the set of protocols available locally. To find a set of protocols which can be used for a data transfer, a communication layer has to intersect the set of locally available protocols with the set of protocols supported by the remote communication layer. The result of this intersection is a possibly empty set of protocols which are supported by both systems involved in the data transfer. If the set contains more than one protocol the decision which particular protocol to use is subject to additional criteria such as certain protocol restrictions (e.g., maximal packet size). Figure 11 depicts the protocol selection process performed by the communication layer. To allow dynamic protocol selection as shown in Figure 11 the communication layer has to determine which protocols are supported by the remote system first. Therefore, both communication layers exchange information about their locally installed protocols before the actual data transfer takes place.

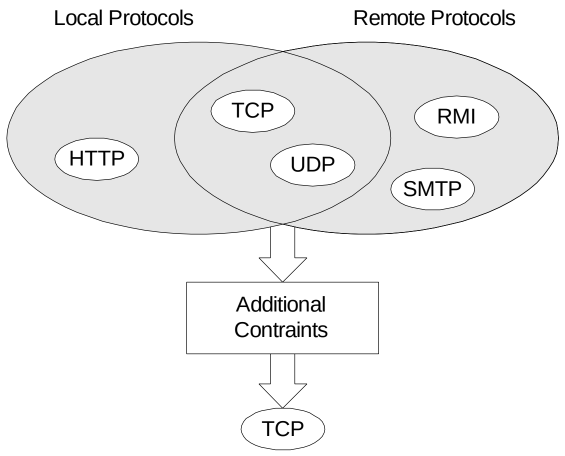

Figure 11: Dynamic protocol selection

The messages and rules involved in the exchange of protocol information prior to the data transfer establish a protocol which is specific to the communication layer and not used for data transfer. The only purpose of this high level protocol is the negotiation of a compatible protocol to be used by both communication layers involved in the data transfer. Figure 12 shows the different steps undertaken by the communication layer to initiate a data transfer with a remote communication layer. Figure 12 also introduces the cache component as a central piece in the communication layer protocol. The cache component allows caching of protocol information and serves as a dictionary for the communication layer. Instead of negotiating a compatible protocol for each data transfer the communication layer uses its cache component to check whether there is a previously negotiated protocol for the intended receiver of data. If this information is existent in the cache component, the data transfer can be started right away without involving any overhead associated with the negotiation of a compatible protocol. If a compatible protocol is not available in the cache component, the communication layer protocol is used to request a list of remotely installed protocols. Based on this list and the protocols installed locally a compatible protocol is selected as described earlier in this section.

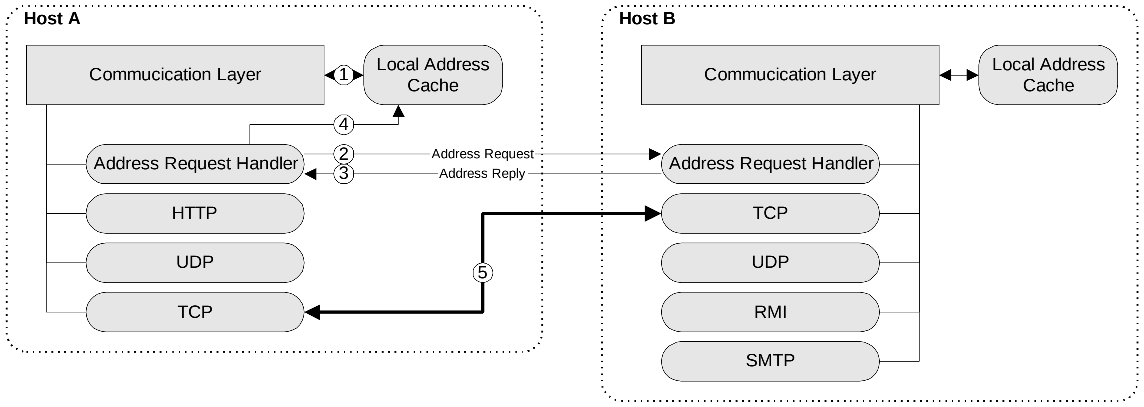

Figure 12: Communication Layer Protocol

The name of the protocol negotiated is then stored in the cache component and can be used for all subsequent communication with the remote location. Protocol information stored in the cache component is invalidated as soon as data transfers fail using the protocol suggested by the cache component. This may happen due to a change in the protocols set installed at the remote location. The invalidation of protocol information eventually forces the communication layer to renegotiate the protocol to be used with the remote location. Thus, by using this caching scheme for protocol information, we reduce the overhead involved in the determination of a compatible protocol but still address the issue of changes in the network architecture and the set of protocols accessible to the system.

## 4.4 Virtual Machine

We mentioned earlier that the main purpose of our system is to provide a framework for mobile Web crawling. Therefore, our approach involves the migration and remote execution of crawlers which are in fact small programs implementing a certain crawling strategy as introduced in section 4.2. In order to be able to send and execute a crawler at a remote location we need an appropriate communication and execution environment installed at the remote location. The design of the communication environment was introduced in section 4.3. This section focuses on facilities needed to allow the actual execution of the crawler code at a remote location. Within the following sections and throughout the text we refer to the crawler runtime environment as a virtual machine.

### *4.4.1. Inference Engine*

In section 4.2 we introduced our rule based approach for the specification of crawler behavior. From the systems point of view, a crawler implementing a certain crawling strategy is represented as a set of rules specified by the user of the system. These rules operate on facts known to the crawler, creating new facts through rule application. The rule application process is implemented by artificial intelligence and expert systems such as CLIPS [GIA97]. Within these systems an inference engine is responsible for determining which rules fire on which facts.

Since the crawlers to be executed within our system are rule based, we can model our runtime environment using an inference engine. To start the execution of a crawler we initialize the inference engine with the rules and facts of the crawler to be executed. Starting the rule application process of the inference engine is equivalent to starting crawler execution. Once the rule application has finished (either because there is no rule which is applicable or due to a external signal), the rules and facts now stored in the inference engine are extracted and stored back in the crawler. Thus, an inference engine establishes a virtual machine with respect to the crawler. Figure 13 depicts the role of the inference engine and summarizes the crawler execution process as described above.

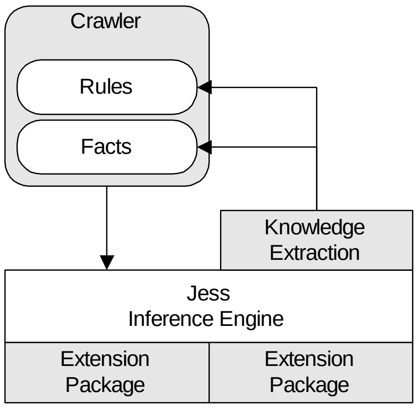

Figure 13: Inference engine and crawler execution process.

Since the implementation of an inference engine is rather complex, we chose to reuse an available engine. Among inference engines that are freely available, we selected the Jess system [FRI97]. Jess is a Java based implementation of the CLIPS expert system [GIA97] which is well known in the artificial intelligence community. Jess does not implement all CLIPS features but does provide most of the functionality necessary for our purposes. The main advantage of using Jess is its platform independence due to its Java based implementation. The fact that our whole system is implemented in Java simplifies the integration of the Jess inference engine into the system architecture significantly. In addition, Jess allows extensions to its core functionality through user implemented packages. This is very important for our system since additional functionality such as Web access is essential for the execution of crawlers and will be used extensively within crawler specifications.

In addition to extending the functionality of Jess through user packages, we had to make modifications to Jess directly. These changes were necessary to allow communication between the crawler and the inference engine. The main problem with the current version of Jess was that we couldn't extract facts from the engine directly. We were able to insert crawler facts and rules into the engine and start the rule application process but there was no direct way to extract the result of the rule application process. Another problem was that Jess was not designed with mobile code in mind. Thus, Jess data structures (e.g., facts) couldn't migrate along with the crawler object. We therefore adapted Jess to allow the serialization of its data structures in order to support crawler mobility.

### *4.4.2. Execution Management*

Section 4.4.1 introduced the core component of our virtual machine architecture, the inference engine. Throughout section 4.4.1 the focus was on how a single inference engine can be used to execute a single crawler. In this section we focus on how the inference engine concept is used within our system to establish a virtual machine for crawlers.

The virtual machine is the heart of our system. It provides an execution service for crawlers that migrate to the location of the virtual machine. As there will only be one virtual machine installed on a single host we expect several crawlers running within the virtual machine at the same time. Therefore, the virtual machine has to be able to manage the execution of multiple crawlers simultaneously. Since the Jess inference engine described in section 4.4.1 can only handle a single crawler, we introduce a special management component which uses multiple inference engines to accommodate the runtime requirements of multiple crawlers. This management component basically establishes an additional level of abstraction above the inference engine level and implements the service the virtual machine has to provide. We refer to this management component as a virtual machine. Figure 14 depicts the management of multiple inference engines through the virtual machine. In addition, Figure 14 shows other system components which the virtual machine cooperates with for the purpose of crawler execution.

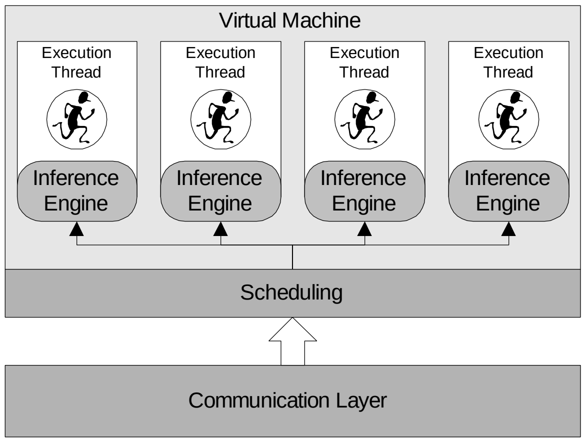

Figure 14: Virtual machine execution management.

Figure 14 depicts the virtual machine as a multithreaded component, maintaining one thread of control for each crawler in execution. Each thread maintains its own Jess inference engine dedicated to the crawler which is executed within the thread. This structure allows several crawlers to be executed in parallel without interference between them because every crawler is running on a separate inference engine. The figure also shows that the virtual machine responds to events issued by the communication layer discussed in section 4.3. The kind of events the virtual machine is listening to are crawler migration events, indicating that a crawler migrated to the location of the virtual machine for execution. The virtual machine responds to these events by scheduling the crawler to the resources managed by the virtual machine.

### *4.4.3. Crawler Scheduling*

The separate scheduling phase preceding the crawler execution as depicted in Figure 14 allows us to specify execution policies for crawlers. Based on the execution policy, the virtual machine can decide whether or not a particular crawler should get executed and when the execution of a crawler should be started. Consider the following examples for crawler execution policies:

**Unrestricted policies:** In case of unrestricted policies, the virtual machine does not restrict crawler execution at all and responds to crawler migration events by allocating the necessary resources (execution thread and inference engine) for the crawler.

**Security based policies:** A security based execution policy allows the system administrator to restrict execution rights to crawlers originating from locations considered to be trustworthy. Thus, a system providing a virtual machine can be protected from crawlers which might be harmful. This policy is useful for creating a closed environment within the Internet called Extranets.

Extranets are basically a set of Intranets connected by the Internet but protected from public access.

**Load based policies:** Load based policies are necessary for controlling the use of system resources (i.e. processor time). For this reason it is desirable to schedule crawler execution during periods when the system load is low and crawler execution does not affect other tasks. To allow this kind of scheduling the system administrator can specify time slots, restricting crawler execution to these slots (e.g., 2-5am). Using a load based policy, the virtual machine responses to crawler migration events by buffering the crawlers for execution until the time specified in the time slot is up.

An appropriate mix of the different policies described above allows the system administrator to control crawler execution in an effective manner. As an example consider a combination of security and load based policies. Using such a combined policy the administrator would be able to specify that mobile crawlers of search engine A should be executed at midnight on Mondays whereas crawlers of search engine B will get their turn Tuesday night. Crawlers of search engines known for their rude crawling would not be accepted at all.

## 4.5 Crawler Manager

Throughout sections 4.3 and 4.4 we introduced our mobile code architecture which is essential for allowing crawler migration and remote crawler execution. In this section and sections 4.6 and 4.7 we focus on the application framework which utilizes the established mobile code architecture to provide a highly customizable information discovery and retrieval framework. We start the discussion of the application framework by introducing the crawler manager whose main purpose is the creation and the management of crawlers. The main tasks of the crawler manager are discussed in sections 4.5.1 through 4.5.3.

### *4.5.1. Crawler Instantiation*

Prior to using a crawler it has to be instantiated by the crawler manager. The instantiation of crawler objects is based on crawler behavior specifications given to the crawler manager. In section 4.2 we introduced our rule based approach for the specification of crawler behavior. Therefore, crawler specifications given to the crawler manager are basically sets of rules. The crawler manager checks those rules for syntactic correctness and assigns the checked rules to a newly created crawler object.

### *4.5.2. Crawler Initialization*

The instantiation of a crawler object by the crawler manager establishes a new crawler carrying a specific crawler program. In addition, we have to initialize crawler objects with some initial facts to begin crawler execution. As an example, consider a crawler which tries to index the Web. This particular kind of crawler will need initial fact seeds containing URL addresses as starting points for the crawling process. The structure and the content of initial facts depends on the particular crawler specification used. Therefore, we can not impose a certain structure for initial facts and have to rely on the crawler programmer to provide facts compatible with the crawler specification.

### *4.5.3. Crawler Migration Management*

In the architecture overview section, we described the architecture framework to be responsible for the creation and the management of mobile code – not for its distributed execution. The execution of mobile code (which are crawler objects in the context of our application framework) is handled by the mobile code architecture introduced in section 4.1.1. The crawler manager is therefore not supposed to start the execution of the initialized crawlers (the application framework does not include a virtual machine anyway). Instead, initialized crawler objects are transferred to a location which provides a crawler runtime environment. Such a location is either the local host (which always has a runtime environment installed) or a remote system which explicitly allows crawler execution through an installed crawler runtime environment. The crawler manager is responsible for the transfer of the crawler to the execution location. The migration of crawler objects and their execution at remote locations implies that crawlers have to return to their home systems once their execution is finished. Thus, the crawler manager has to wait for returning crawlers and has to indicate their arrival to components interested in the results which the crawlers carry such as the query engine.

To fulfill the tasks specified above, the crawler manager uses an inbox/outbox structure similar to an email application. Newly created crawlers are stored in the outbox prior to their transmission to remote locations. The inbox buffers crawlers which have returned from remote locations together with the results they contain. Other system components such as the query engine discussed in section 4.6 can access the crawlers stored in the inbox through the crawler manager interface. Figure 15 summarizes the architecture of the crawler manager.

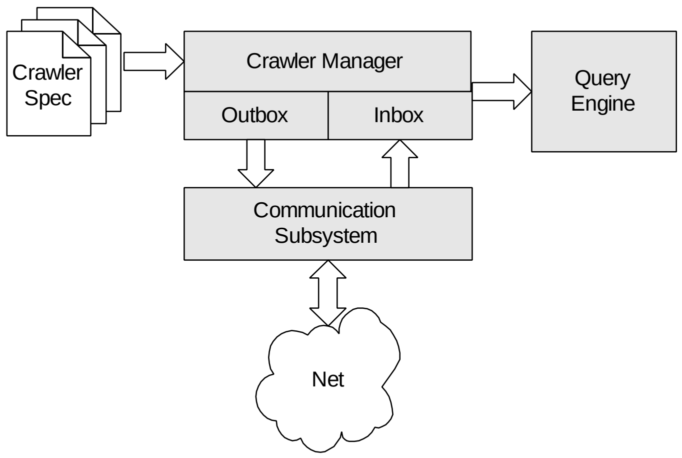

Figure 15: Crawler manager architecture.

Figure 15 indicates the tight cooperation of the communication subsystem and the crawler manager. Although we introduced the communication subsystem as part of the crawler runtime environment in section 4.3, we reuse this subsystem within our application framework. The reason for this is that the crawler manager needs to communicate with remote crawler runtime environments in order to perform crawler transfers. Therefore, we have to integrate a compatible communication facility in our application framework. The easiest way to ensure compatible communication is to use the communication subsystem of the crawler runtime environment as the communication facility of the application framework.

## 4.6 Query Engine

With the crawler manager discussed in section 4.5, our application framework is able to create and manage crawler objects which perform information retrieval tasks specified by the user of the system. Although our application framework is responsible for crawler creation and management, the meaning of a crawler program remains transparent with respect to the application framework. In particular, the structure and the semantics of facts used to represent retrieved information within the crawler is not known to the application framework. The semantic and fact structure is determined by the user of the framework who specifies the crawler program using the rule based approach introduced in section 4.2.

The fact that neither the structure nor the semantic of the information retrieved by the crawler is known to the application framework raises the question of how the user should access this information in a structured manner. In particular, we need to develop an approach which allows the user to exploit his knowledge about the crawler program when it comes to accessing the information retrieved by the crawler. Clearly this approach can not be implemented statically within our framework. We instead provide a query engine which allows the user to issue queries to crawlers managed by the framework. This way, the user of the framework can encode its knowledge about the information structure and semantics within a query which gets issued to the crawler. Thus, by implementing a query engine we support the dynamic reasoning about the information retrieved by a crawler without imposing a static structure or predefined semantics for the information itself. The query engine basically establishes an interface between the application framework and the application specific part of the system. From a more abstract point of view, the query engine establishes a SQL like query interface for the Web by allowing users to issue queries to crawlers containing portions of the Web. Since retrieved Web pages are represented as facts within the crawler memory, the combination of mobile crawlers and the query engine provides a translation of Web pages into a format that is queriable by SQL. Sections 4.6.1 through 4.6.3 introduce important aspects of the query engine and give examples of how the engine is used within our information retrieval application.

### *4.6.1. Query Language*

The idea of a query engine as an interface between our framework and the user application implies that we have to provide an effective and powerful notation for query specification. Since the design of query languages has been studied extensively in the context of relational database systems, we decided to use a very limited subset of SQL (Structured Query Language) for our purposes. The language subset implemented by our query engine is currently limited to simple SQL select statements (no nesting of statements and a limited set of operators). Although this limits the functionality of our query language significantly, it seems to be sufficient for our purposes. To illustrate this consider the following example. Assume a user created a crawler which indexes Web pages in order to establish an index of the Web. Also assume also that the following fact structure is used to represent the retrieved pages.

- *URL:* The URL address of a page encoded as a string.
- *Content:* The source code of a page as a string.
- *Status:* The HTTP status code returned for this page as an integer.

Table 3 shows some example queries, the user might need in order to get the page data from the crawler in a structured manner.

Table 3: Query language examples.

| ***Operation*** | ***Query*** |
|---|---|
| Find successfully loaded pages in order to add them to the index. | SELECT url, content<br>FROM Page p<br>WHERE p.status = 200 |
| Find dead links in order to delete them from the index. | SELECT url<br>FROM Page p<br>WHERE p.status = 401 |

### *4.6.2. Query Execution*

Based on the query language introduced in section 4.6.1 the user of our framework has a notation to express what kind of data he wants to be extracted from a crawler. The execution of queries is the responsibility of the query engine which is part of our framework and serves as the interface between the framework and the user (or his application). Within this section we discuss how user queries are transformed and executed by the query engine.

By recalling the basic idea behind crawler execution as discussed in section 4.4 we realize that any data a mobile crawler retrieves or generates is represented as facts within the crawler. These facts are carried back by the crawler to the system the crawler was created at. Thus, the data which user queries are executed upon is basically a collection of facts extracted from a crawler object. The task of the query engine can now be stated as finding those facts (or just the certain fact attributes of interest) which match with the query issued by the user.

By further recalling that an inference engine allows us to reason about facts by providing rules, we realize that we can exploit the inference engine mechanism to implement our query engine. Instead of using a classical database oriented approach like relation algebra to execute queries, we transform a query into a rule and use an inference engine to let this query rule operate upon the collection of facts extracted from the crawler. Figure 16 depicts the query engine architecture which results from the observation made so far.

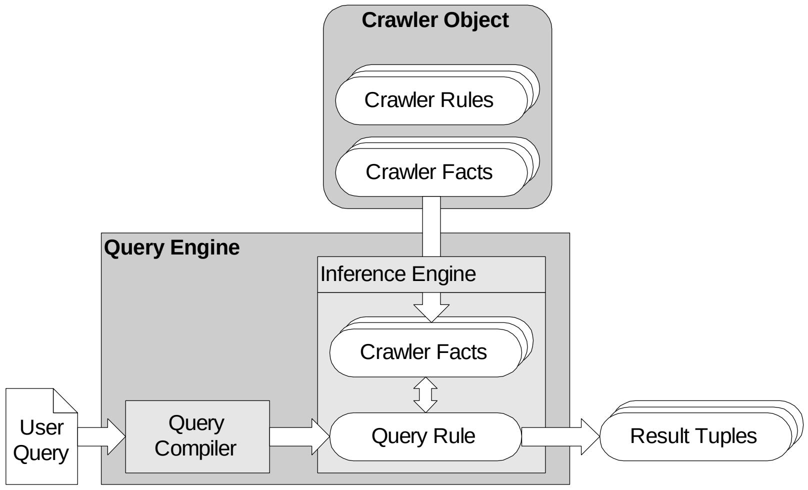

Figure 16: Query engine architecture.

By using this architecture we get our query engine almost for free because the information to be queried is already given as facts and can be extracted from crawler objects easily. We already have all components needed for the inference engine because we used the same approach when we implemented the virtual machine as discussed in section 4.4. The only thing needing specification is how a query should be transformed into a rule to match and extract the desired information based on the given facts.

Fortunately, the transformation mentioned above is straightforward. Recall, that the rule structure of the inference engine allows us to define a set of patterns which need to match with certain facts before the rule can fire. By referencing facts within the rule pattern, we require the specified facts to be existent within the fact base before the rule can fire. This models a selection mechanism within the inference engine which is powerful enough to implement the query language we defined in section 4.6.1. As an example for a query transformation consider the following example.

Table 4: Query transformation example.

| ***Query:*** | ***Rule:*** |
|---|---|
| `SELECT url,fulltext`<br>`FROM Page` | `(defrule Query`<br>` (Page (url ?url) (fulltext ?fulltext))`<br>` →`<br>` (assert (QUERY_RESULT url ?url fulltext ?fulltext)))` |

The simple query on the left hand side of Table 4 is supposed to retrieve the two specified attributes of all facts named *Page*. Please note that the structure of the *Page*-facts could be arbitrarily complex; the query would just retrieve the specified attributes. The right hand side of Table 4 shows how this simple query can be represented as a rule within the inference engine. The query rule has a pattern which requires that a *Page*-fact is present in the fact base before the rule can fire. The pattern further references the attributes specified in the query and keeps their value in local variables. With this pattern, the rule will fire each time the inference engine can find a fact named Page with the two specified attributes. Thus, the inference engine does the work of selecting matching facts for us – we just tell the inference engine how a matching fact looks like and what to do with it.

In case the rule fires it executes the assert statement which will create a new result fact containing the attribute name and values specified in the query. By creating result facts, it is relatively easy to identify the result of a query once the rule application process has stopped. The union of all created result facts establishes the result that the original query was supposed to compute.

### *4.6.3. Query Results*

In order to use the query result within other system components, we have to extract the facts generated by the query rule from the inference engine first. Since all result facts have a common static name (QUERY_RESULT) we could extract them based on this name information. This approach requires us to look at the names of all facts sequentially which is not feasible for a very large fact base. Instead we make use of the fact that the inference engine assigns a unique identifier to each fact inserted into the engine. These identifiers are merely integers which are consumed in nondecreasing order. By recalling the way the query engine operates we realize that the fact base upon which the query is going to operate is inserted into the inference engine first. It is only after the insertion of the fact base that the query rule can create any result facts. Thus, fact identifiers for query result facts are always strictly greater than the fact identifier of the last fact inserted before the query was issued. We can use this property to extract the query result quickly by using the fact identifiers rather than the fact names.

Let $x$ be the number of facts before the query execution and $y$ be the number of facts after query execution respectively. Due to the sequential ordering of facts within the inference engine we can get the query result by extracting exactly $y$-$x$ facts starting with identifier $x$.

By using the scheme described above we get a set of query result facts containing the requested information. Due to the inconvenience involved with the representation of data as inference engine facts, we would like to transform the query result into another, more abstract representation. Since our query language introduced in section 4.6.1 is a limited database query language, we derive our result representation directly from the scheme commonly used in relational databases. Any data stored or retrieved from a relational database is represented as a flat table. These tables consist of a (possibly empty) set of tuples where each tuple establishes one row of the table. Since the structure and syntax of our query language suggests that the user should expect the result to be something close to a relational database result, we represent query result facts as a set of tuples. In our approach, each tuple has a name (i.e. the name of the table in the relational database context) and a set of attribute name/value pairs (the column names and their associated values). This representation makes it easier for other system components to work with the result of a query because it establishes a level of abstraction above the plain result facts retrieved by the query engine.

## 4.7 Archive Manager

Within the last couple of sections we introduced the components and approaches used to retrieve information using mobile crawlers. The framework architecture established so far can be utilized to retrieve an application specific data set from the Web. As an example consider a subject specific search engine which would utilize mobile crawlers to index Web pages containing information about health care. Although such a specialized search engine does not need to retrieve the whole Web content, a considerably large amount of data is retrieved by mobile crawlers. The analysis of the retrieved data is an application specific issue to be addressed by algorithms provided by the user. Therefore, we do not provide any components which deal with the information retrieved by mobile crawlers other than the query engine introduced in section 4.6. Instead, we provide a flexible data storage component, the archive manager, which allows the user to define how the retrieved data should be stored to accommodate analysis carried out later on by the user application. This approach separates the data retrieval architecture provided by our framework from the user specific application dealing with the retrieved data. By allowing the user to specify how the retrieved data should be stored, we achieve a high degree of independence between our framework and the user application. More importantly, with this approach we do not impose any particular organization for the retrieved data which allows the user to introduce highly customized data models.

In order to be able to store large amounts of information by still providing a customizable and structured access to the stored data, we decided to use a relational database (i.e., Sybase) as the back-end storage system. Within the next sections we introduce components built on top of the relational database which allow the user to specify how to store the retrieved data using a relational database management system.

### *4.7.1. Database Connection Manager*

The discussion above suggests that we have to incorporate database access into our framework. To be as flexible as possible, we have to provide transparent access to possibly multiple databases distributed within a local network.

Since our framework is based on Java we have decided to use the JDBC (Java Database Connectivity [HAM97]) interface to implement the necessary database mechanisms. JDBC provides a standard SQL interface to a wide range of relational database management systems by defining Java classes which represent database connections, SQL statements, result sets, database metadata, etc. The JDBC API allows us to issue SQL statements to a database and process the results that the database returns. The JDBC implementation is based on a driver manager that can support multiple drivers to allow connections to different databases. These JDBC drivers can either be written in Java entirely or they can be implemented using native methods to bridge existing database access libraries. In our particular case, we use a pure Java driver provided by Sybase [SYB97] to access the Sybase database installed in our department. The JDBC configuration used for our framework as depicted in Figure 17 uses a client side JDBC driver to access relational databases. Therefore, the Sybase server process does not need to be configured in a special way in order to provide JDBC database access for our framework. The JDBC API uses a connection paradigm to access the actual databases. Once a database is identified by the user, JDBC creates a connection object which handles all further communication with the database.

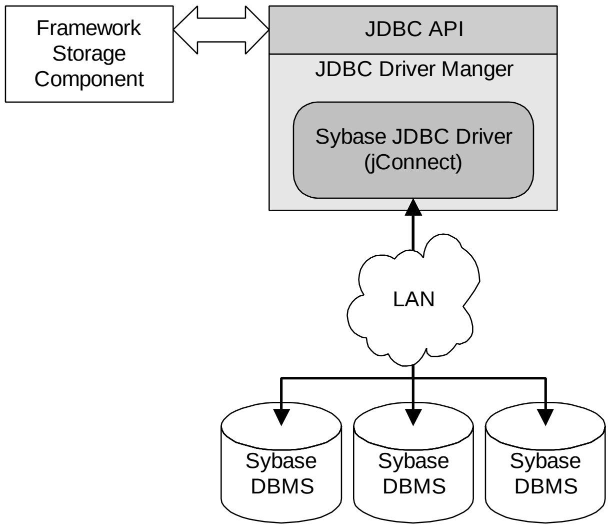

Figure 17: JDBC Database access to Sybase DBMS.

Since our archive manager has to provide transparent access to several databases at the same time, we need to design a connection manager that manages the different database connections on behalf of the archive manager. The archive manager can then issue commands to different databases without being concerned about the underlying JDBC connections. The main task of the connection manager is therefore to establish database connections (i.e. to initiate the creation of the corresponding JDBC connection objects) on demand and to serve as a repository for JDBC database connection objects. By assigning a unique name to each connection, the archive manager can access different databases at the same time without the need to establish a connection to the database explicitly. Figure 18 summarizes the architecture of the connection manager and its relationship with the archive manager.

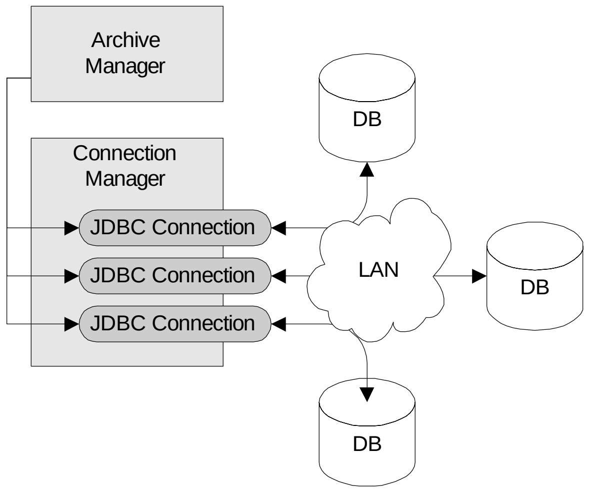

Figure 18: Connection manager operation.

### 4.7.2. *Database Command Manager*

The connection manager as described in the previous section allows the archive manager to connect to multiple, distributed databases within the local network. By establishing a database connection through the connection manager, the archive manager can interact with the database by issuing SQL commands. Such an interaction requires the archive manager to have knowledge of the structure and the semantics of each database it works with. As outlined earlier in this section, we do not intend to impose any particular data model and storage structures upon the user of our framework. Thus, the organization of data with the database cannot be known to the archive manager because it is defined in the context of the user application which utilizes our framework. To solve this problem, we have introduced an additional layer of abstraction between the user context (i.e. the database schema) and the framework context (i.e. the archive manager).

This abstraction is implemented by the database command manager which provides a mapping between the archive manager and the database schema. This mapping basically establishes a database specific vocabulary which is used to interact with certain parts of the database. Specifically, the database command manager provides a set of abstract database command objects which each implement an operation specific to the underlying database scheme (e.g., insert page command, delete page command). Thus, each database command object embeds all the lower level SQL functionality (i.e. the JDBC code to issue SQL commands) needed to provide a higher level operation which has a reasonable granularity with respect to the database scheme. Figure 19 depicts this approach and clarifies the relationship between command objects, command manager, database management system and database scheme.

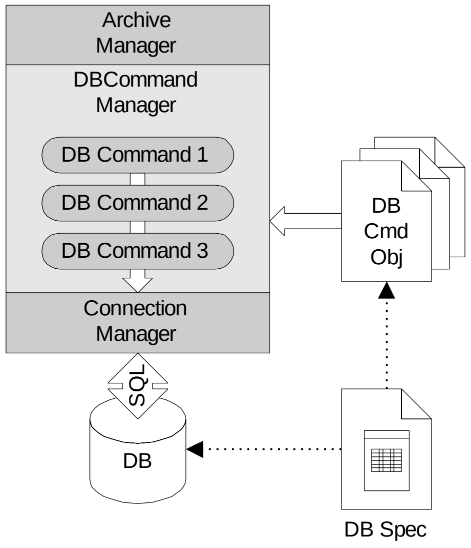

Figure 19: Database command manager architecture.

Figure 19 shows how the database command manager serves as a layer of abstraction between the database and the archive manager accessing the database. The database command manager loads abstract database commands dynamically at runtime to provide a database specific vocabulary to the archive manager. One can see the abstract command objects as a type of database access plug-in, providing database specific functionality to our application framework architecture. It is the responsibility of the framework user to provide appropriate database command objects embedding the necessary database scheme specific code. By providing database command objects, the framework user configures the framework to work with the application specific database scheme.

Consider for example a user who wants to retrieve and store health care specific pages from the Web. After defining crawlers which retrieve the actual information the user has to design a database scheme which is appropriate to store the retrieved data. Suppose the database scheme is simple and consists of two tables, one for the URL address of a page and one for the actual page attributes. To make this particular database design known to the archive manager, the user provides a database command object called *AddPage* which contains the actual SQL code necessary to insert the Web page data into both tables. By loading the *AddPage* command object into the command manager, it becomes part of the vocabulary of the particular database. Since all storage operations of the archive manager are based on the abstract database vocabulary, there is no need for our framework (i.e. the archive manager) to know anything about the underlying database scheme it uses. The structure of the scheme as well as the semantics of its attributes are embedded in the command objects. Figure 20 depicts the example using SQL pseudocode for command object implementation. The approach described in the example is also suitable for handling changes to the underlying database schema. In case the user wants to extend the database scheme to accommodate some new requirements (e.g., add a new relational table or modify the attributes of an existing one), all he has to do is to change the existing database command objects or introduce new ones. Since all new and changed command objects can be loaded by the database command manager at runtime, we do not need to recompile any components of our framework to accommodate the new database scheme.

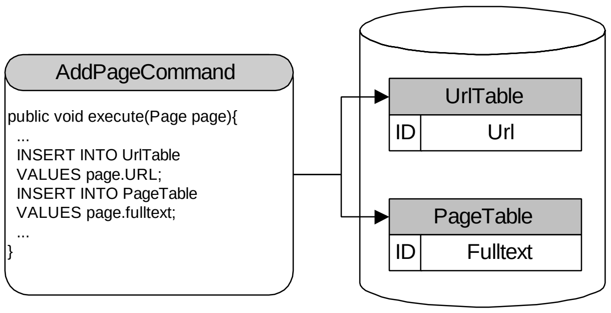
```
public void execute(Page page){
  ...
  INSERT INTO UrlTable
  VALUES page.URL;
  INSERT INTO PageTable
  VALUES page.fulltext;
  ...
}
```

Figure 20: Database command object example.

### *4.7.3. Archive Manager Operations*

In the last two sections we introduced basic components needed for the operation of the archive manager. Section 4.7.1 introduced database connection manager and JDBC for transparent access to relational database management systems. Section 4.7.2 introduced the database command manager which allows the archive manager to use a dynamically loaded database specific vocabulary rather than plain SQL commands to interact with the database. Both sections discussed the database part of the archive manager primarily. In this section we focus on how the archive manager utilizes the established database subsystem to store the actual data retrieved by mobile crawlers.

There are two main issues which remain to be addressed by the archive manager. First, the archive manager needs to know what kind of data must be extracted from a given crawler object. Second, the archive manager needs to know how it is supposed to store the data extracted from a crawler. By looking at the component architecture discussed so far, we realize that these two issues are already addressed by two particular components:

**Query engine:** The issue of how to extract data from a given crawler object is addressed by the query engine introduced in section 4.6. By using the query engine, the archive manager can specify what data it wants to be extracted by stating specific criteria the extracted data must match.

**Database command manager:** The issue of how to store extracted information is addressed by the database command manager in conjunction with database command objects as introduced in section 4.7.2. By using the database specific vocabulary provided by the database command manager, the archive manager can store extracted data in the database without being concerned about the particular database implementation (i.e. the database scheme used).

Since the archive manager is going to use the query engine to extract data from a given crawler object, it needs to have insight into the data the particular crawler carries. Unfortunately, the archive manager does not have direct access to the data inside the crawler because structure and semantics of the data retrieved by the crawler is specified by the user. Thus, only the crawler programmer knows which particular data structure contains the data which needs to be stored in the database. Consequently, only the crawler programmer will be able to provide the set of queries which will extract the appropriate data from the crawler. Thus, the archive manager has to provide a mechanism which allows the user of the framework to associate each crawler with a corresponding set of queries to be used to extract data retrieved by particular crawler. Based on such associations, the archive manager can determine what queries should be issued as soon as a particular crawler finishes execution.

The execution of queries by the query engine will result in a set of tuples which have been introduced in section 4.6.3. To store the result tuples in the database, the archive manager needs one or more data command objects. Therefore, we need to find a way to specify which commands objects the archive manager is supposed to use for a particular query/crawler combination. Since this problem is very similar to the one we solved before, we can use the same association based mechanism again. To implement this, the archive manager provides a mechanism which allows the user to register a database command object to be responsible for handling the result of a query issued to a particular crawler. Based on this information, the archive manager can determine which database command object(s) should be used to store the query result in the database.

The archive manager supports the specification of associations between crawlers, queries and database command objects by introducing storage rules. A single storage rule keeps track of all necessary association for one particular crawler. Thus, the user of the system has to specify one storage rule for each crawler he is going to use. Storage rules can be added to the archive manager dynamically to allow easy configuration of the system. At runtime, the archive manager selects the appropriate storage rule based on the crawler and processes the crawler in accordance to the rule. Figure 21 summarizes the purpose and the application of storage rules within the archive manager.

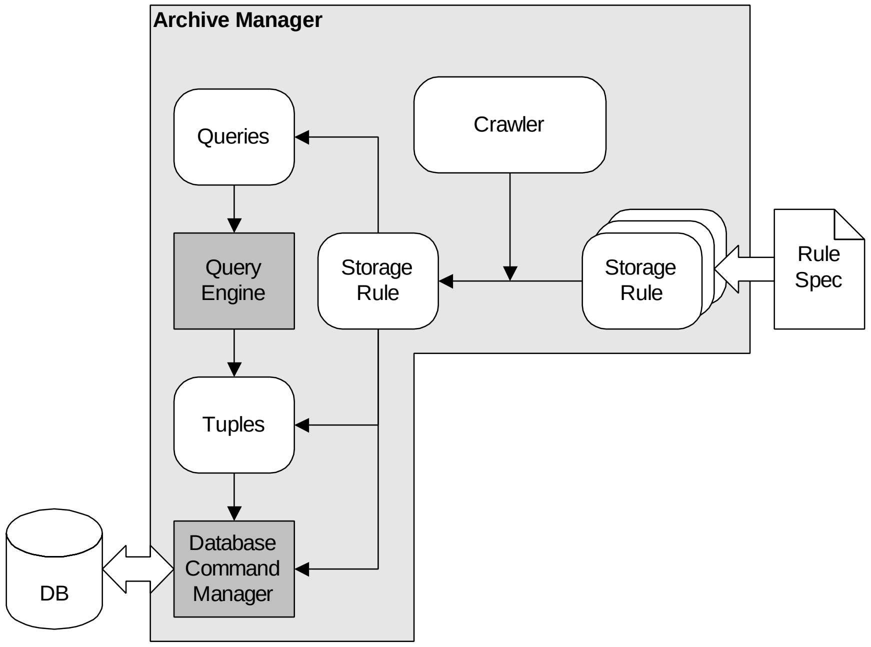

Figure 21: Storage rule application in the archive manager.

## 5. PERFORMANCE EVALUATION

### 5.1 Evaluation Configuration

The goal of this performance evaluation is to establish the superiority of mobile Web crawling over the traditional crawling approach. Our evaluation focuses on the application of mobile crawling techniques in the context of specialized search engines which cover certain subject areas only. Beside measurements specific to mobile crawlers in the context of specialized search engines, we provide material which strongly suggests that mobile Web crawling is beneficial for general purpose search engines as well.

For the performance evaluation of our mobile crawling architecture we established a system configuration which allows us to evaluate mobile crawling as well as traditional crawling within the same environment. By doing so, we ensure, that the results measured for the different approaches are comparable with each other. By looking at the general architecture of the mobile crawler approach, we recall that the main difference between our mobile crawlers and the traditional approach is the location where the crawlers operate. A mobile crawler operates close to the data it needs to access, whereas a traditional crawler ignores the distribution of data sources and operates from a home base via remote access. This suggests that we can simulate traditional crawling techniques using a mobile crawler which does not move around. In that case, none of the optimizations due to mobility discussed in section 3.1 are applicable. In particular, we realize that a mobile crawler and a traditional crawler running the same crawling algorithm on the same machine will produce identical results. Thus, we can measure the improvements in Web crawling due to mobility by using two identical mobile crawlers in the following way: One of the crawlers simulates a traditional crawler by accessing data through the communication network. The other one migrates to the data source first, taking advantage of local data access and the other mobility depended optimizations as described in section 3.2. Figure 22 depicts the corresponding evaluation configuration in terms of the system components introduced in section 4.1.

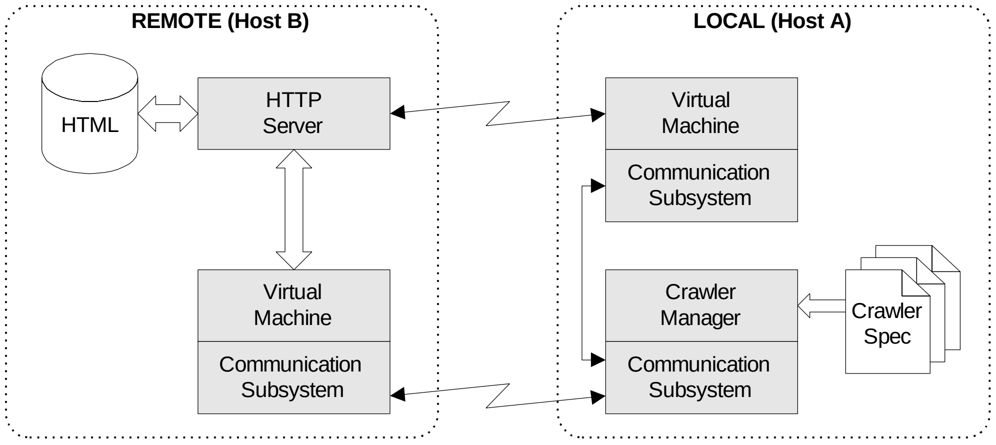

Figure 22: Evaluation configuration.

Our experiment shown in Figure 22 is set up as follows. We install two virtual machines at two different locations (Host A and B) within the network. This allows us to execute a mobile crawler locally (Host B) as well as remotely (Host A) with respect to the data source (i.e. a Web server). We simulate a traditional crawler by running a mobile crawler on the virtual machine installed on host A. This generates statistics for traditional (non mobile) Web crawling because a crawler running on host A has to access the

HTTP server installed on host B remotely. To get statistics for mobile crawling, we then let the same crawler migrate to host B, where it can take advantage of local access to the Web server. To ensure that the statistics generated with the configuration described above are comparable, we have to impose some additional constraints on several system components.

**Network:** To get accurate network load data, we have to make sure that our measurements are not affected by other network activity. Therefore, we run our experiments on a dedicated point to point dialup connection guarenteeing that we do not have to compete for network bandwidth with other applications.

**HTTP server:** In order to analyze the benefits of local versus remote data access we have to make sure that the performance characteristics of the targeted HTTP server do not change during our experiments. In particular, the server load due to HTTP request messages should only depend on the traffic generated by our test crawlers. We enforce this constraint by installing our own HTTP server which is only accessible within our experimental testbed.

**HTML data set:** The different crawler runs conducted for our analysis have to operate upon identical HTML data sets in order to provide accurate results. Furthermore, the HTML data set used should be a representative subset of the Web with respect to critical page parameters such as page size and number of links per page. We address this issue by using a considerably large set of Web pages as the data set to be used by the HTTP server. In this context static means, that neither the HTTP server nor the HTML pages contain any dynamic elements (e.g., database queries, active server pages) which could alter the page content.

**Crawler specification:** As described earlier, our evaluation configuration uses identical crawler algorithms for the mobile crawling run as well as for the traditional crawling run. In addition to this, we have to specify our crawlers such that their behavior is reproducible in order to ensure consistent measurements for identical crawling algorithms. This means that a crawler always has to access the same set of pages in the same order as long as data set does not change. This constraint excludes us from using crawling algorithms with randomized crawling strategies. Unless stated otherwise, we use a straightforward breadth-first fulltext crawler for our experiments and do not make use of . We deliberately do not use advanced crawling strategies as described by Cho [CHO97] in order to keep our results general.

## 5.2 Benefits of Remote Page Selection

Remote page selection as introduced in section 3.2.2 allows a mobile crawler to determine whether or not a certain page is relevant in a subject specific context. Due to crawler mobility, this decision can be made right at the data source which avoids the transmission of possibly irrelevant information over the network. This suggests that the network load can be seen as a function of the degree of remote page selection. We expect this function to establish a nearly linear dependency. One end of this linear spectrum is established by comprehensive crawlers which cannot benefit from remote page selection at all because they select and index every page. For highly subject specific crawlers, at the other extreme, we expect significant reductions in network load because only a possibly small number of Web pages have to be transmitted.

Since the benefit of remote page selection depends on the subject area a particular crawler is interested in, it is hard to measure the effects in an experimental setup which only approximates the actual Web environment. Despite this difficulty, we tried to get realistic measurements by installing a highly subject specific data set at the HTTP server. The advantage of a specialized data set is that we can use our advance knowledge of the data set characteristics to devise crawlers which have a varying degree of overlap with the subject area. As data set for our experiments we used the Java programming tutorial which is a set of HTML pages (about 9Mbyte total) dealing with Java programming issues. Based on our knowledge of the content of these documents, we derived different sets of keywords which served as selection constraints to be enforced by our crawlers. Our experimental crawling algorithm was then modified such that a crawler would only index pages which contain at least one of the keywords given in the keyword set. With this approach, we were able to adjust the overlap between crawler subject area and data set subject area by providing different keyword sets. For our measurements we used the following different crawlers.

**Stationary crawler (S1)**: This is a crawler running on host A of our evaluation configuration. S1 therefore simulates a traditional crawler since it has to access the data source remotely by sending HTTP request messages over the network. Due to the remote data access, the assigned keyword set is irrelevant since all pages need to be downloaded before S1 can analyze them. Therefore, the network load caused by S1 is independent of the keyword set.

**Mobile Crawler (M1)**: This is a mobile crawler which migrates to host B of our evaluation setup to take advantage of local access to the Web pages being crawled. M1 has a keyword set such that it considers all Web pages as being relevant. Therefore, M1 downloads as many pages as S1 does.

**Mobile Crawler (M2 to M4)**: These crawlers are identical to M1 but use different keyword sets. The keyword sets have been chosen such that the overlap between crawler subject area and Web page content decreases for each crawler. Therefore, M2 retrieves less pages than M1, M3 retrieves less than M2, and M4 retrieves less than M3.

For each crawler we determined the network load with and without compression applied to the transmitted data. Figure 23 summarizes our measurements for the first 100 pages of our data set.

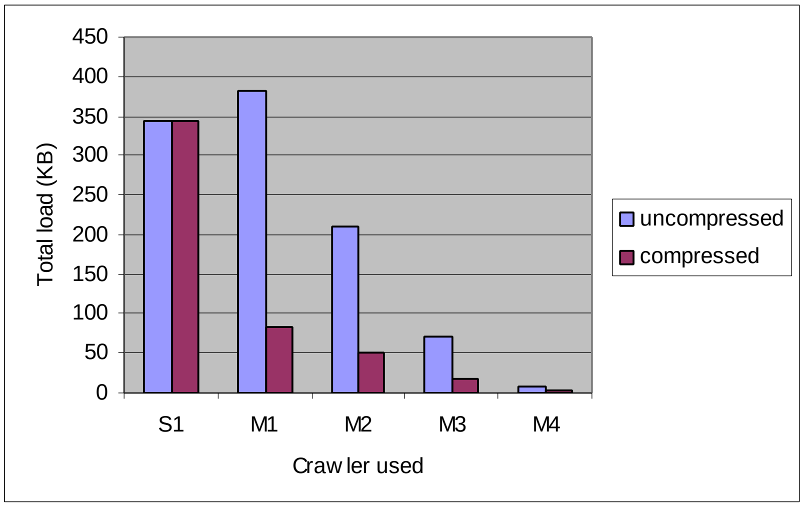

Figure 23: Benefits of remote page selection.

Our reference data in Figure 23 is the network load caused by the tradition crawler S1 because this crawler can neither use remote page selection nor remote page compression to reduce the amount of data being transmitted. Figure 23 also shows that the network load caused by M1 is slightly higher than the one caused by the traditional crawler S1 if we do not allow page compression. This is due to the overhead of crawler migration. If we allow M1 to compress the pages before transmitting them back, M1 outperforms S1 by a factor of 4. The remaining bars in Figure 23 show the results for mobile crawlers M2 to M4. These crawlers use remote page selection to reduce the number of pages to be transmitted over the network based on the assigned keyword set. Therefore, M2, M3, and M4 simulate subject specific Web crawling as required by subject specific search engines.

### 5.3 Benefits of Remote Page Filtering

Measuring the benefits of remote page filtering is even more difficult than measuring the effects of remote page selection. Since different applications are likely to focus on different aspects of HTML pages, crawlers have to preserve a variable amount of page data in order to represent a Web page sufficiently. The amount of data needed for page representation can vary significantly from several thousand to less than one hundred bytes. One extreme are fulltext crawlers which use the complete source code of a HTML page, the other extreme is established by crawlers which represent a Web page with its URL address only.

To measure the actual benefits of remote page filtering we modified our crawler algorithm such that only a certain percentage of the retrieved page content is transmitted over the network. By adjusting the percentage of page data preserved by the crawler, we can simulate different classes of applications. Figure 24 summarizes our measurements for a static set of 50 HTML pages. Each bar in Figure 24 indicates the network load caused by our mobile crawler M1 depending on the filter degree assigned to the crawler. The network load is measured relative to the network load of our traditional crawler S1.

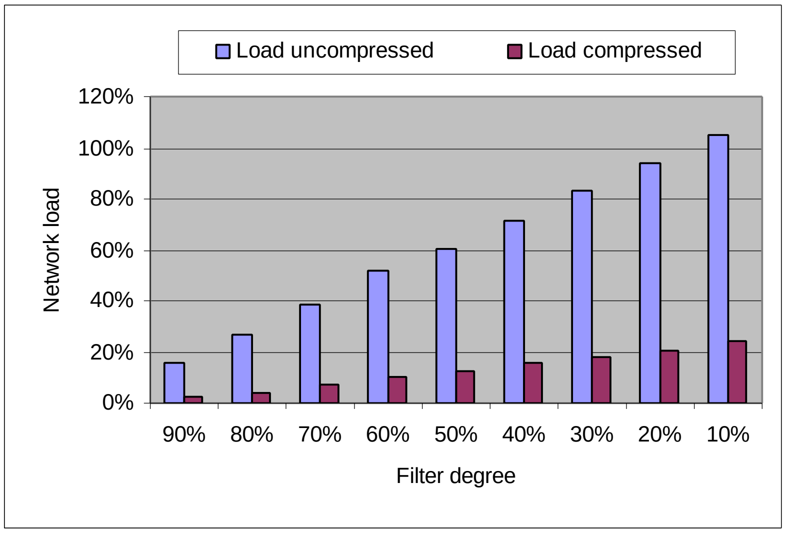

Figure 24: Benefits of remote page filtering.

Since a traditional crawler can not take advantage of remote page filtering, S1 creates a network load of 100% independent of the filter degree. The measurements depicted in Figure 24 suggest that remote page filtering is especially useful for crawlers which do not use remote page compression. The benefit of remote page filtering is less significant if page compression is applied too. For example, a filter degree increase of 10% results in a reduction of transmitted data by 2.5% only if data compression is combined with page filtering. Note the high (above 100%) network load for the very last measurement. If we filter out less than 10% of a each Web page (this preserves 90% or more of the page) and do not use page compression, we actually increase the amount of data transmitted over the network. This is due to the overhead of transmitting the crawler to the remote location and depends on the amount of data crawled. The more data the crawler retrieves, the less significant the crawler transmission overhead will be.

### 5.4 Benefits of Page Compression

The benefits of remote page selection and filtering as discussed in sections 5.2 and 5.3 depend heavily upon assumptions about the data being crawled (i.e. certain subject areas) and the page representations being used. In this section we shift our focus to the benefits of remote page compression, a technique which is always applicable for mobile crawlers without any further assumption about the context a particular crawler is operating in. Therefore, remote page compression can be used completely independent of all other techniques discussed before. This fact makes remote page compression especially interesting for comprehensive fulltext crawlers (needed for general purpose search engines) which cannot benefit from remote page selection and filtering. Page compression can only be beneficial in cases where a mobile crawler operates at, or at least close to the location of the data source (i.e. HTTP server). This allows the crawler to compress the retrieved data **before** transmitting it to the home system. Thus, as all other techniques discussed before, remote page compression requires a crawler to migrate to the data source.

Since Web pages are basically human readable ASCII messages, we can use well known text compression algorithms such as gzip for Web page compression. These compression algorithms perform extremely well when used on large homogeneous data sets. Since a set of Web pages retrieved by a mobile crawler establishes such a large homogeneous data set, we expect the benefits of remote page compression to be significant. Figure 25 summarizes the results measured with and without remote page compression activated within the crawler configuration.

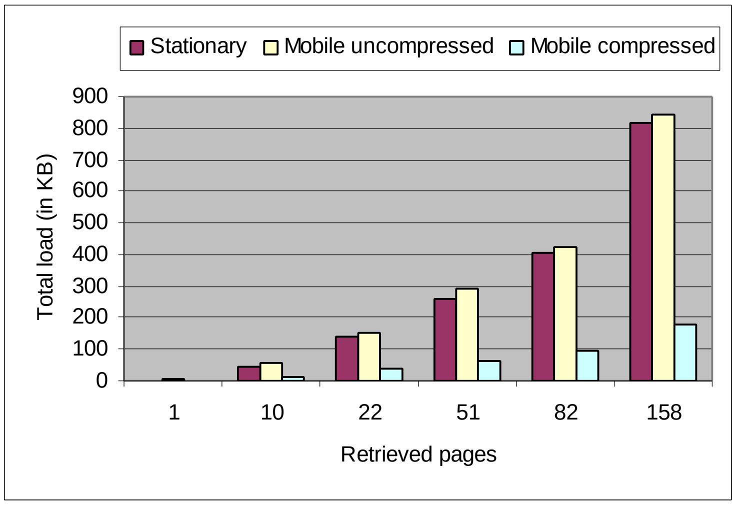

Figure 25: Benefits of remote page compression.

Figure 25 shows the total network load caused by three different crawlers with respect to the number of pages retrieved. For each measurement, we doubled the number of pages each crawler had to retrieve. Since we focused on the effects of page compression, all other crawler features such as remote page selection were turned off.

As in section 5.2, we find that mobile crawling without page compression and traditional crawling perform similar with respect to the network load. Mobile crawling without page compression involves a little overhead due to crawler migration. As soon as a mobile crawler compresses pages before transmitting them back, we can see a significant saving in network bandwidth. The data presented in Figure 25 suggests that mobile crawlers achieve an average compression ratio of 1:4.5 for the Web pages they retrieve. Since the communication network is the bottleneck in Web crawling, mobile crawlers could work about 4 times faster than traditional crawlers because it takes less time to transmit the smaller amount of data.

# 6. Conclusion

## 6.1 Summary

This thesis introduced an alternative approach to Web crawling based on mobile crawlers. The proposed approach surpasses the centralized architecture of the current Web crawling systems by distributing the data retrieval process within the network. In particular, using mobile crawlers we are able to perform remote operations such as data analysis and data compression right at the data source before the data is transmitted over the network. This allows for more intelligent crawling techniques and especially addresses the needs of applications which are interested in certain subsets of the available data only.

We developed an application framework which implements our mobile Web crawling approach and allows user applications to take advantage of mobile crawling. In the context of our framework, we introduced a rule based approach to crawler behavior specification which provides a flexible and powerful notation for the description of crawler behavior. This rule based notation allows a user to specify what a mobile crawler should do and how it should respond to certain events.

The performance results of our approach are very promising. Mobile crawlers can reduce the network load caused by crawlers significantly by reducing the amount of data transferred over the network. Mobile crawlers achieve this reduction in network traffic by performing data analysis and data compression at the data source. Therefore, mobile crawlers transmit only relevant information in compressed form over the network.

## 6.2 Future Work

The prototype implementation of our mobile crawler framework provides an initial step towards mobile Web crawling. We identified several issues which need to be address by further research before mobile crawling can be used in a larger scale.

The first issue which needs to be addressed is security. Crawler migration and remote execution of code causes severe security problems because a mobile crawler might contain harmful code. We argue that further research should focus on a security oriented design of the mobile crawler virtual machine. We suggest introducing an identification mechanism for mobile crawlers based on digital signatures. Based on this crawler identification scheme a system administrator would be able to grant execution permission to certain crawlers only, excluding crawlers from unknown (and therefore unsafe) sources. In addition to this, the virtual machine needs to be secured such that crawlers cannot get access to critical system resources. This is already implemented in part due to the execution of mobile crawlers within the Jess inference engine. By restricting the functionality of the Jess inference engine, a secure sandbox scheme (similar to Java) can be implemented easily.

A second important research issue is the integration of the mobile crawler virtual machine into the Web. The availability of a mobile crawler virtual machine on as many Web servers as possible is crucial for the effectiveness of mobile crawling. For this reason, we argue that an effort should be spend to integrate the mobile crawler virtual machine directly into current Web servers. This can be done with Java Servlets which extend Web server functionality with special Java programs.

A third issue is research in mobile crawling algorithms. None of the current crawling algorithms have been designed with crawler mobility in mind. For this reason, it seems worthwhile to spend some effort in the development of new algorithms which take advantage of crawler mobility. In particular these algorithms have to deal with the loss of centralized control over the crawling process due to crawler mobility.

## References

[BEL97] BellCore, Netsizer – Internet Growth Statistics Tool, Bell Communication Research, 1997 (http://www.netsizer.com)

[BER96] Berners-Lee, T., Hypertext Transfer Protocol – HTTP/1.0, RFC 1945, Network Working Group, 1996

[BOW95] Bowman, C. M., Danzig, P. B., Hardy, D. R., Manber, U., Schwartz, M. F., Wessels, D. P., Harvest: A Scalable, Customizable Discovery and Access System, Technical Report, University of Colorado, Boulder, Colorado, USA, 1995

[BRI97] Brin, S., Page, L., The Anatomy of a Large-Scale Hypertextual Web Search Engine, Computer Science Department, Stanford University, Stanford, CA, USA, 1997

[CHO97] Cho, J., Garcia-Molina, H., Page, L., Efficient Crawling Through URL Ordering, Computer Science Department, Stanford University, Stanford, CA, USA, 1997

[FIN94] Finin, T., Labrou, Y., Mayfield, J., KQML as an agent communication language, Computer Science Department, University of Maryland Baltimore County, Baltimore, MD, USA, 1994

[FRI97] Friedman-Hill, E., Jess Manual, Sandia National Laboratories, Livermore, CA, USA, 1997

[GIA97] Giarratano, J. C., CLIPS User's Guide, Software Technology Branch, NASA/Lyndon B. Johnson Space Center, USA, 1997

[GOS96] Gosling, J., McGilton, H., The Java Language Environment, White Paper, Sun Microsystems, Mountain View, CA, USA, 1996

[HAM97] Hamilton, G., Cattell, R., JDBC: A Java SQL API, White Paper, Sun Microsystem, Mountain View, CA, USA, 1997

[HAR96] Harrison, C.G., Chess, D.M., Kershenbaum, A., Mobile Agents: Are they a good idea?, T.J. Watson Research Center, IBM Research Division, NY, USA, 1996

[KAH96] Kahle, B., Archiving the Internet, Scientific American, 03/1996 http://www.archive.org/sciam_article.html

[KOS93] Koster, M., Guidelines for Robot Writers, Web document, 1993 http://info.webcrawler.com/mak/projects/robots/guidelines.html

[KOS95] Koster, M., Robots in the Web: threat or treat?, Web document, 1995 http://info.webcrawler.com/mak/projects/robots/threat-or-treat.html

[KOS96] Koster, M., A Method for Web Robots Control, Informational Internet Draft, Network Working Group, 1996

[KOS97] Koster, M., The Web Robots Pages, Web document, 1997 http://info.webcrawler.com/mak/projects/robots/robots.html

[MAE94] Maes, P., Modeling Adaptive Autonomous Agents, MIT Media-Laboritory, Cambridge, MA, USA, 1994

[MAE95] Maes, P., Intelligent Software, Scientific American 273 (3), September, 1995

[MAU97] Mauldin, M. L., Lycos Design Choices in an Internet Search Service, IEEE Expert Interview, 1997 http://www.computer.org/pubs/expert/1997/trends/x1008/mauldin.htm

[MCB94] McBryan, O. A., GENVL and WWWW: Tools for Taming the Web, First International Conference on the World Wide Web, CERN, Geneva, Switzerland, May 25-27, 1994

[NWA96] Nwana, H. S., Software Agents, An Overview, Knowledge Engineering Review, Vol. 11(3), Oct./Nov. 1996, Cambridge University Press, 1996

[PIN94] Pinkerton, B., Finding What People Want: Experience with the WebCrawler, The Second International WWW Conference Chicago, USA, October 17-20, 1994

[SUL98] Sullivan, D., Search Engine Watch, Mecklermedia, 1998 http://www.searchenginewatch.com

[SYB97] Sybase, Inc., jConnect for JDBC, Technical White Paper, Sybase, Inc., Emeryville, CA, USA, 1997 http://www.sybase.com/products/internet/jconnect/jdbcwpaper.html

[WHI96] White, J., Mobile Agent White Paper, General Magic, Sunnyvale, CA, USA, 1996 http://www.genmagic.com/agents/Whitepaper/whitepaper.html

[WOO95] Wooldridge, M., Intelligent Agents: Theory and Practice, Knowledge Engineering Review, Vol. 10(2), June 1995, Cambridge University Press, 1995