Budapesti Műszaki és Gazdaságtudományi Egyetem
Budapest University of Technology and Economics

Tudományos Diákköri Konferencia

Crawler.NET

A component-based distributed framework for web traversal

Crawler.NET: Komponensalapú elosztott keretrendszer a web bejárására

Hunyadi Levente
Pallos Péter

Konzulens:
Gincsai Gábor, doktorandusz,
Automatizálási és Alkalmazott Informatikai Tanszék

2006. november 17.

Kivonat

Az internet napjaink egyik meghatározó információforrása, azonban a segítségével fellelhető információ szórt formában, nem feltétlenül eredeti tárolási struktúrájának megfelelően van jelen, hanem egymással hivatkozásokon keresztül összekapcsolt, eltérő terminológiájú dokumentumok formájában érhető el. Gyakori feladat, hogy egy-egy keresőkérdésnek megfelelő dokumentumhalmazt adjunk vissza válaszként. A keresés ilyen megvalósításhoz azonban feltétlenül szükséges a weben található tartalom ütemezett bejárása és feltérképezése.

Alkalmazásunk célja olyan hagyományos személyi számítógépeken futó elosztott rendszer kialakítása, amely hatékonyan végzi a weboldalak letöltését, az azokban található hivatkozások kinyerését és követését, illetve az ezen bejárással nyert irányított gráf tárolását. Egy elosztott rendszer számos előnyt élvez egy központosított rendszerrel összevetve a skálázhatóság, a hálózati terheléselosztás és -csökkentés terén az elosztottság miatti többletkomplexitás és kommunikációs nehézség ellenére. Rendszerünk egyesíti az elosztott rendszerből eredő hatékonyságnövelést és az egyszerű kezelhetőséget.

Alkalmazásunk egymáshoz lazán kapcsolódó, ám jól definiált interfészekkel illeszkedő, üzenetekkel kommunikáló egységek összességeként épül fel. A laza csatolás egyrészt biztosítja, hogy párhuzamosítható folyamatokat akár dinamikusan változtatható számú feldolgozóegység végezhessen, másrészt lehetővé teszi, hogy a rendszerbe új, előre nem látható funkcionalitás legyen beépíthető. A rendszer kezdeti képe deklaratív módon, XML állományok segítségével határozható meg, amely alapján az elosztott alkalmazás automatikusan felépíthető. A laza csatolású rendszer révén az egyes egységek a rendelkezésre álló erőforrások – processzor, merevlemez, hálózati sávszélesség – és a webkiszolgálók minél kisebb mértékű igénybevételére, a beérkező adat rugalmas és hibatűrő feldolgozására, illetve a működés során előforduló hibákból a minél kisebb veszteség mellett történő helyreállásra helyezhetik a hangsúlyt.

Alkalmazásunk megvalósítása a .NET keretrendszerre alapul, erőteljesen támaszkodik annak Base Class Library (BCL) részében megvalósított szálkezelési és távoli kommunikációbeli építőelemeire. A rendszer forráskódja C# és Mercury programozási nyelvű, mindkettő a .NET által használt köztes kódra (Intermediate Language) fordítható. Az adatműveleteket a flexibilitás érdekében adatbázis-kiszolgáló végzi, amely területen a Microsoft SQL Server megvalósítására esett a választás, ez azonban a későbbi megvalósítások során a hatékonyság további fokozása érdekében hasonló illesztőfelület mellett egyszerű struktúrált állományra cserélhető, alkalmazásunk csak csekély mértékben használja ki az SQL erejében rejlő lehetőségeket.

Abstract

The Internet is one of today's primary information sources yet information available through this medium is scattered and is not necessarily present in its original storage format. In fact, information is available as a set of interconnected documents each with a possibly different terminology. Nevertheless, it is a typical task to return a set of documents that match a given query, exemplified by the wide-spread use of web search engines. Such retrieval of documents, however, is only possible through a periodic traversal of the Web.

The goal of our application is to create a distributed system running on a network of conventional PCs that supports downloading documents, extraction of hyperlinks from these documents, following the extracted hyperlinks and storing the resultant directed graph. A distributed system has several advantages over a centralized system in terms of scalability, network load dispersion and load reduction despite the additional complexity and communication overhead it produces. Therefore, the system we propose couples the efficiency of a distributed architecture with easy manageability.

Our application consists of a set of loosely coupled units with well-defined interfaces that communicate with messages. On one hand, loose coupling caters for concurrent execution of parallelizable processes by a possibly dynamically variable number of processing units. On the other, it enables easy extension of the current system with future functionality. The initial setup of the system is declaratively configurable by means of XML files based on which the distributed application is automatically initialized. Through the loosely coupled architecture, actual components can focus on the smallest possible extra demand on network resources and target flexible and error-tolerant transformation of incoming data and recovery from critical failures with the least possible loss.

Our application is intended for the .NET platform and fully utilizes thread management and remote communication facilities the Base Class Library of the platform provides. The source code of the platform has been implemented in C# and Mercury, both compile to the Intermediate Language used by the .NET framework. For the sake of flexibility, data operations are performed by a relational database. Though our particular implementation uses Microsoft's SQL Server, this can be replaced with a structured file in later stages of development for further efficiency gain as our implementation moderately exploits the power SQL offers.

Contents

Chapter 1

Introduction

Since its birth in 1993, the Internet has undergone a tremendous change. In particular, it has dramatically increased in size and its content has reached an unprecedented diversity. Meanwhile, locating useful information has become a cumbersome endeavor. Hence, the role of automated *search engines* to find relevant information has increased substantially.

Despite its heterogeneity, the Internet has retained its most basic structure of interconnected documents. In this sense, the term *document* can refer to any kind of text or multimedia content, such as static or dynamically generated HTML pages, photo images, PDF files, word processor documents, etc. In order to make documents eligible to be returned as search results, their content should be parsed by search engines and relevant information extracted to compile an *index*. User search *queries* are then evaluated against this index.

In order to compile their massive set of document indices, search engines rely on *web crawlers*. Web crawlers traverse the Internet by hopping from document to document following *hyperlinks* that can be extracted from each. The operation of a web crawler can be described by the simple algorithm in Figure 1.1. [29]

Albeit the algorithm appears simple, an efficient implementation of a web crawler is not straightforward. Data structures, such as the hyperlink set, usually hold millions of records all of which do not simultaneously fit into memory in their naive representation. Data has to be stored either in a space-efficient manner or partially read from and written to disk possibly augmented with a intermediary cache. Despite their huge size, data structures have to be accessed quickly, which can be complex for more sophisticated search strategies. Even a simple breadth-first traversal strategy requires specialized approach to speedily determine the next page to download for a given host.

In addition to space- and time-efficient data structures, downloading and parsing documents should be performed by multiple agents running in parallel not to be limited by network latency or response time. Concurrent agents should be coordinated and data

1. The crawler is *seeded* with an initial set of hyperlinks (URLs). Practically, this set contains references to documents of high relevance, such as general directories or specialized directories organized around a specific topic.
2. A hyperlink is chosen (and removed) from the set according to some traversal strategy. This can be a simple breadth- or depth-first or a focused crawling strategy based on some concept of relevance (e.g. for a search engine specialized in scientific articles such documents have a greater weight and are ranked with a more prominent priority).
3. The document related to the hyperlink is downloaded.
4. The document is parsed for any hyperlinks it may contain.
5. The new hyperlinks are filtered against various crawling criteria. Some crawlers may choose not to follow certain types of hyperlinks, e.g. those that contain query strings (name-value parameters in the `?` part of the URL, such as `page=index` in `http://example.com?page=index`).
6. Hyperlinks not discarded in step 5 are appended to the hyperlink set. Steps from 2 are repeated with the extended set.
7. The iteration terminates if the hyperlink set becomes empty or the crawler is forcefully stopped.

Figure 1.1: The basic web crawler algorithm.

passed efficiently between them. Moreover, if the application is designed to scale, cooperation of distributed agents on remote machines and their sharing common data are both crucial issues.

1.1 Scope and structure of this paper

Even though web crawlers are widely employed in general or focused search engines and in web content and web structure mining [12], details of many of these systems are unknown or descriptions are too terse to allow reproducibility, partly to avoid malicious site creators deceiving engines to gain higher result rankings. In addition, systems in published literature either have a centralized architecture or do not lend themselves to easy configuration. Our goal is to design a web crawler that harnesses the scalability potential in distributed systems whilst providing ample support for pluggable components and configurable behavior.

In this paper, the authors present a novel architecture for implementing a distributed web crawler. The proposed system comprises of independent *components,* each of which

is running on a different thread. The components are loosely coupled by means of local or remote FIFO *queues* (or *connectors*). Both components and connectors are fully object-oriented with their corresponding base classes providing fundamental services common to all inheritors, especially with respect to initialization and coordination. Each component encapsulates a specific crawler task, such as scheduling download requests to servers, downloading or parsing documents. Connectors transmit data between components and thus allow transparent distribution of components among distributed machines.

With the component architecture at their disposal, the authors outline a full-fledged distributed web crawler with a central coordinator. The coordinator (the *server*) is responsible for assigning tasks to agents (*clients*), which perform the tasks of collecting and analyzing web pages, and building the URL reference graph. Both the server and clients are distributable across multiple machines and themselves consist of highly configurable components.

Web crawlers realized as part of research are implemented in C, C++, Java, Perl or Python and no known distributed crawler is available for the .NET platform. The versatility of this platform without a severe speed penalty have led the authors to choose this framework for implementing their design.

This paper is structured as follows. Chapter 2 gives an introduction into concepts relevant on the field of web crawling including traversal strategies and download policy. Related work, namely the Mercator, PolyBot and UbiCrawler crawling systems, is discussed in Chapter 3. The basic component architecture underlying our crawler is presented in Chapter 4. Chapter 5 outlines our system and elaborates on the exact components that actually perform the crawl. The paper concludes with Chapter 6, which evaluates the performance of the crawler both analytically and dynamically, and sketches future work.

1.2 Own contributions and intended audience

The design and the .NET implementation of the component framework described in detail in Chapter 4 are entirely our own contribution. This in particular includes but is not limited to automatized thread management for components, transparent local and remote asynchronous message-based communication and the freely configurable pluggable component model. Similarly, the components explained in Chapter 5 are our own work, though some of these are based on algorithms described in Chapter 2. In addition, for the sake of higher efficiency, several disk-resident but memory-cached data structures and network management classes have been realized as opposed to re-using the basic all-purpose collection types provided by the .NET framework, which perform poorly in the particular

large-data, high-load scenario.

Crawler.NET is an open-source project under constant development, hosted on SourceForge.net [8] under the UNIX name `webcrawler`. The project is available for download through anonymous CVS login.

This paper builds on an intermediate knowledge of application-level web protocols, most emphatically HTTP [20], and SGML languages like HTML [26] or XML [14]. Throughout the paper but especially in Chapter 4 and 5, we make full use of the object-oriented design paradigm including inheritance and interfaces. Additionally, some programming experience in version 2.0 of the .NET platform and the C# language is assumed to fully comprehend code examples. In particular, the paper will use the concepts of generic interfaces, classes and collections or .NET attributes (annotations) without further clarification.

Chapter 2

Background

In this chapter, various complementary tasks are introduced that are essential in implementing a web crawler though did not directly appear in the crawling algorithm in Chapter 1. This includes traversal strategies, operation synchronicity, download policy, content parsing, the hyperlink filter and traversal constraints. Traversal strategies define the principle according to which the next hyperlink to download is chosen. Operations, related to retrieving documents in particular, may be performed synchronously or asynchronously, the two cases will be compared. Download policy aims to control the aggressiveness of the crawler so that no single host should be bombarded with so many requests it becomes incapable of generating replies. In the content parsing phase, transfer encodings are reversed, the code page of the HTTP reply is discovered, and finally hyperlinks are extracted and resolved into absolute links. The hyperlink filter helps discard hazardous links following which the crawler can be easily lured into the so-called crawler traps. Traversal constraints give further partially external help to avoid visiting pages with no valuable or other malicious content. The concluding section of this chapter discusses URL and host locality, which is essential to the operation of our crawler.

2.1 Traversal strategies

In the majority of cases, crawlers are a front-end to search engine indexers. Indexers *tokenize* documents downloaded by crawlers and extract relevant keywords, which are then inserted into their database. [31] For the database to remain effective, it is essential that the crawler download important pages and re-visit quickly changing pages more frequently. [29]

2.1.1 Breadth-first traversal

A wide-spread yet simple heuristics to download important pages in the initial phase of the crawl is to use *breadth-first traversal* (also known as *breadth-first search* or BFS). If documents are treated as nodes and hyperlinks as edges of a graph, breadth-first traversal can be seen as the advance of a wavefront in the graph (Figure 2.1). The algorithm starts with a single node behind the wavefront. In the next phase, the wavefront engulfs all neighbors of the initial node. In each phase, further nodes directly accessible through edges from nodes just behind the wavefront are engulfed. We define the *parent* of a node as the node from which it was first accessed as the wavefront proceeded. Nodes accessed during the traversal, their respective parents and the edges that run between them constitute the *spanning tree* of the breadth-first traversal (Figure 2.3). The algorithm is illustrated in Figure 2.2. [19]

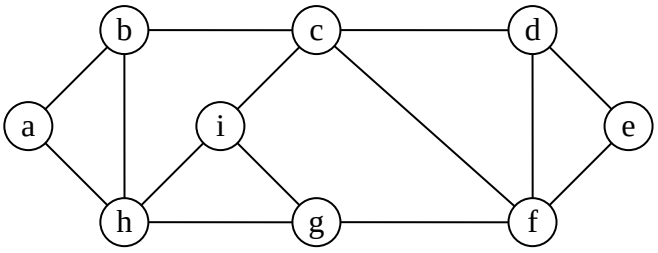

Figure 2.1: Example graph in which the different traversal algorithms are illustrated.

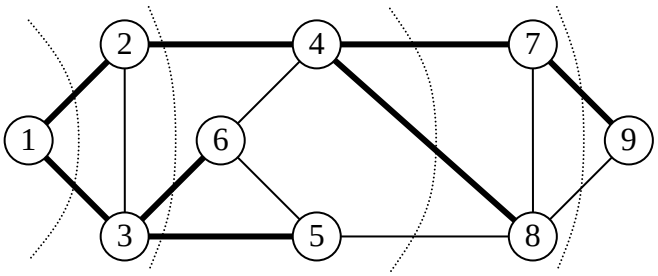

Figure 2.2: Expansion order of nodes and wavefronts in the breadth-first traversal algorithm.

2.1.1.1 Implementation issues

The breadth-first strategy is commonly implemented by means of a FIFO queue (Figure 2.4). As they are visited, neighboring nodes are appended to the end of a data structure that can conceptually be seen as a list. Whenever free capacities allow, nodes to visit are extracted from the head of the list.

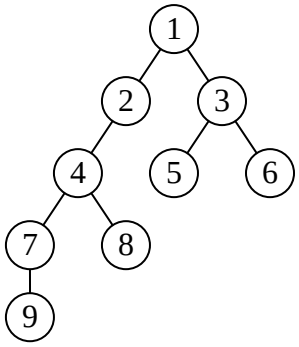

Figure 2.3: Breadth-first spanning tree.

```
procedure bfs(a)
    enqueue(q, a)
    mark v as visited
    while q is not empty
        v := dequeue(q)
        process v
        for all unvisited vertices v' adjacent to v
            mark v' as visited
            parent[v'] := v
            enqueue(q, v')
```

Figure 2.4: Pseudocode of the breadth-first traversal algorithm implemented by means of a FIFO queue. q denotes the queue and a the node of the graph with which the traversal is initiated.

Besides simplicity, the primary advantage of the above implementation is easy generalization to multi-threaded environments. Threads can simultaneously extract nodes from the head of the list and append unvisited neighboring nodes to the end. To prevent that multiple threads visit a single node in parallel during execution, testing if a node has been visited and marking it as such should be an atomic (indivisible) operation.

Unfortunately, the property that every node is accessed on the shortest possible path does not inherit to a multi-threaded environment. Constraints such as downloading a portion of the Internet at a distance of k clicks from a given starting page are not easily satisfied. The traversal does not guarantee that that documents at a maximum distance of k from the initial node will also be at a maximum distance of k in the spanning tree corresponding to the actual crawl.

2.1.1.2 Application

In our particular scenario, namely web traversal, documents are represented by their hyperlinks. A crawler implementation will store URLs in its FIFO queues and visiting a node

will correspond to downloading the document. Neighboring nodes can be seen as hyperlinks the document contains. Lest documents are retrieved multiple times, hyperlinks a document contains are checked against a *hyperlink seen structure* as they are extracted. The hyperlink seen structure may be a huge store of links to visited documents (as in the above discussion) or a more sophisticated probabilistic data structure that determines with sufficient probability if a link has already been seen.

A remarkable variant of the breadth-first strategy is host-specific traversal. In this scenario, once a hyperlink to a document on a given host is discovered, the crawler follows hyperlinks until all documents from the host are retrieved. The primary benefit of this approach is that the hyperlink seen data structure is confined to a host, hence it is significantly smaller in size and can be discarded once the host-specific traversal is complete. The primary drawback is that it does not necessarily download important pages first and isolated "islands" on hosts may be missed.

2.1.1.3 Quality assessment

Experience shows that a breadth-first strategy tends to download important pages first. The assumption is based on two notable properties. First, web sites expose relevant information on their main pages and less valuable information is usually found "deeper" as traversed from the main page. Second, if a document is relevant, the probability of a hyperlink on the web to this very document is much higher, therefore the document is likely to be discovered early in the crawl.

The seed of a breadth-first traversal strategy is often initialized with hyperlinks to popular *directories*. Directories are web sites of hierarchically organized collections of hyperlinks to various topics of interest. They are often maintained directly by people, which guarantees that the documents they reference are excellent sources of information on their field. [29] If a breadth-first traversal algorithm is seeded with a list of directories, it is even more likely that relevant pages are discovered without much delay.

Nevertheless, a breadth-first strategy may not be adequate to quickly find and traverse sites centered around a special topic. In *focused crawling,* the next document to download is decided based on a relevance function rather than on a pure first come first serve principle. The function guides the crawler as to which hyperlinks to process with higher priority.

2.1.2 Depth-first traversal

In depth-first traversal, the selection strategy for the next node is different that what is applied in breadth-first traversal. While breadth-first traversal proceeds with the imme-

diate neighborhood of the current node, depth-first traversal digs deep into the graph. More precisely, the next node the algorithm chooses is always a neighbor of the current node that has not already been visited. Whenever we run out of such neighbors, the current node is changed to its parent. The order of the nodes as traversed by the algorithm is seen in Figure 2.5.

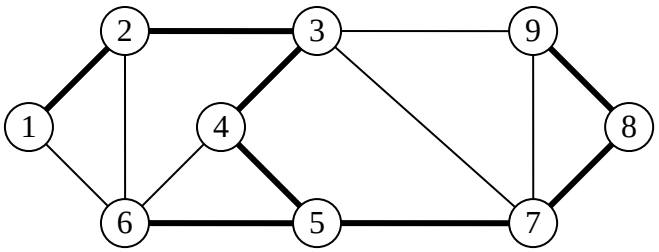

Figure 2.5: Expansion order of nodes in the depth-first traversal algorithm.

2.1.2.1 Implementation issues

Depth-first traversal has two different implementations. The recursive implementation (Figure 2.6) is hard to generalize to a multi-threaded environment but an iterative approach (Figure 2.7) allows simultaneous access by means of a shared stack. [19]

```
procedure dfs-recursive(v)
    process(v)
    mark v as visited
    for all vertices i adjacent to v not visited
        parent[i] := v
        dfs-recursive(i)
```

Figure 2.6: Pseudocode of the recursive depth-first traversal algorithm.

The primary advantage of depth-first traversal is that the order of retrieved documents resembles the way people browse the web. This can trick web servers that try to determine from the order of page requests whether they are dealing with an automatic crawler or a real visitor. While this may come handy in particular situations, the algorithm has a very serious drawback, namely, it is likely to be captured by crawler traps (see Section 2.5).

2.2 Call synchronicity

The efficiency of a web crawler depends largely on its ability to conduct multiple simultaneous (most commonly data retrieval) operations. Two major ways exist as to how an

```
procedure dfs(a)
    push(q, a)
    mark v as visited
    while q is not empty
        v := pop(q)
        process v
        for all unvisited vertices v' adjacent to v
            mark v' as visited
            parent[v'] := v
            push(q, v')
```

Figure 2.7: Pseudocode of the iterative depth-first traversal algorithm.

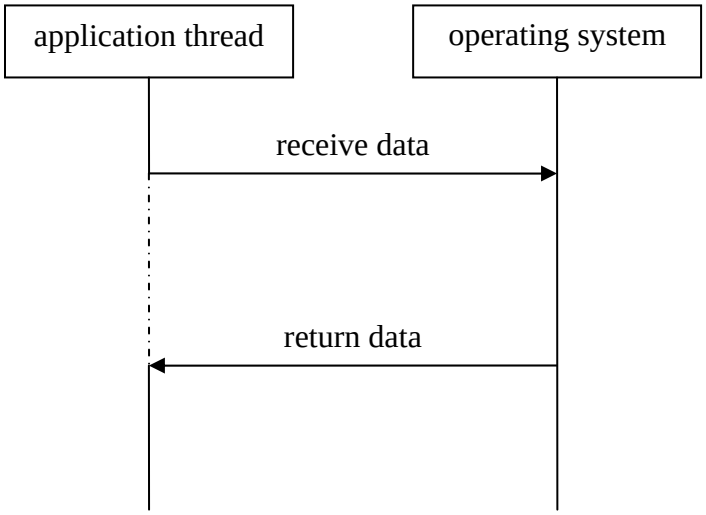

Figure 2.8: A synchronous data transfer. Dashed lines indicate idle thread state. Unlike in the asynchronous case, the initiator thread is suspended and returns with available data once it is awaken.

operation may be carried out.

- In the more common *synchronous call model,* after an operation is initiated, execution blocks until the response (most commonly some form of data) is available or a timeout expires. Meanwhile, the initiating thread remains idle and can perform no useful task. (Figure 2.8) Despite this seems a considerable loss of computing capacity, in a multi-threaded environment task parallelization largely counterbalances this effect. While the thread waits for the operation to complete, others that would perform intensive computation can be scheduled by the operation system.

  In general, the synchronous call model leads to cleaner system design as regards data structure management because manipulation on a set of data is confined to a specific thread. In an object-oriented language this often means that class data

members are accessed from the single thread that is executing class methods only, which does not necessitate delicate locking mechanisms.

- As the name suggests, in the case of the *asynchronous call model,* the executing thread does not block after initiating an operation. In contrast, execution resumes immediately. Though not always the case, a notification may be requested about the completion of the operation by means of a *callback function*, which is registered upon initiating the operation. Once the operation is ready, the callback function is run, which may read pending data from the input stream. If data is not yet available at the time of the data retrieval request, as is often the case when the request method is called from the initiator thread, the caller thread is suspended as in the synchronous case. (Figure 2.9)

  The asynchronous call model usually gives more freedom to the programmer to control operation behavior in detail with respect to timings but it is significantly more difficult to track in complex applications. In particular, data structures are accessed from multiple threads and locking mechanisms are required to avoid data corruption.

The synchronous and asynchronous call models are not fundamentally different concepts. In fact, the synchronous model is a special case in which the synchronous statement is substituted by an asynchronous statement and a *thread join.* In a thread join, the executing thread pauses until some condition becomes true. The condition is either the expiry of a timeout or the completion of the callback function registered with the asynchronous statement, whichever happens first.

2.3 Download policy

In order to retrieve documents, the crawler, acting as a client, *connects* to a remote server (host). After a successful connection, it may read data from and write data to the so-called *network streams*, which are abstractions representing the data flow through the connection. Using either the asynchronous model or a multi-threaded synchronous model, a crawler should open a large number of concurrent network streams to remain effective. If a large proportion of these connections is directed against a single host, a denial of service may occur, especially for smaller servers or dynamic content generated on the fly using interpreted languages.

Some traversal strategies are prone to lead to denial of service if not properly tackled. For instance, in a breadth-first traversal strategy, URLs extracted from a document are appended to the end of the FIFO queue. Let us suppose that 100 URLs were extracted

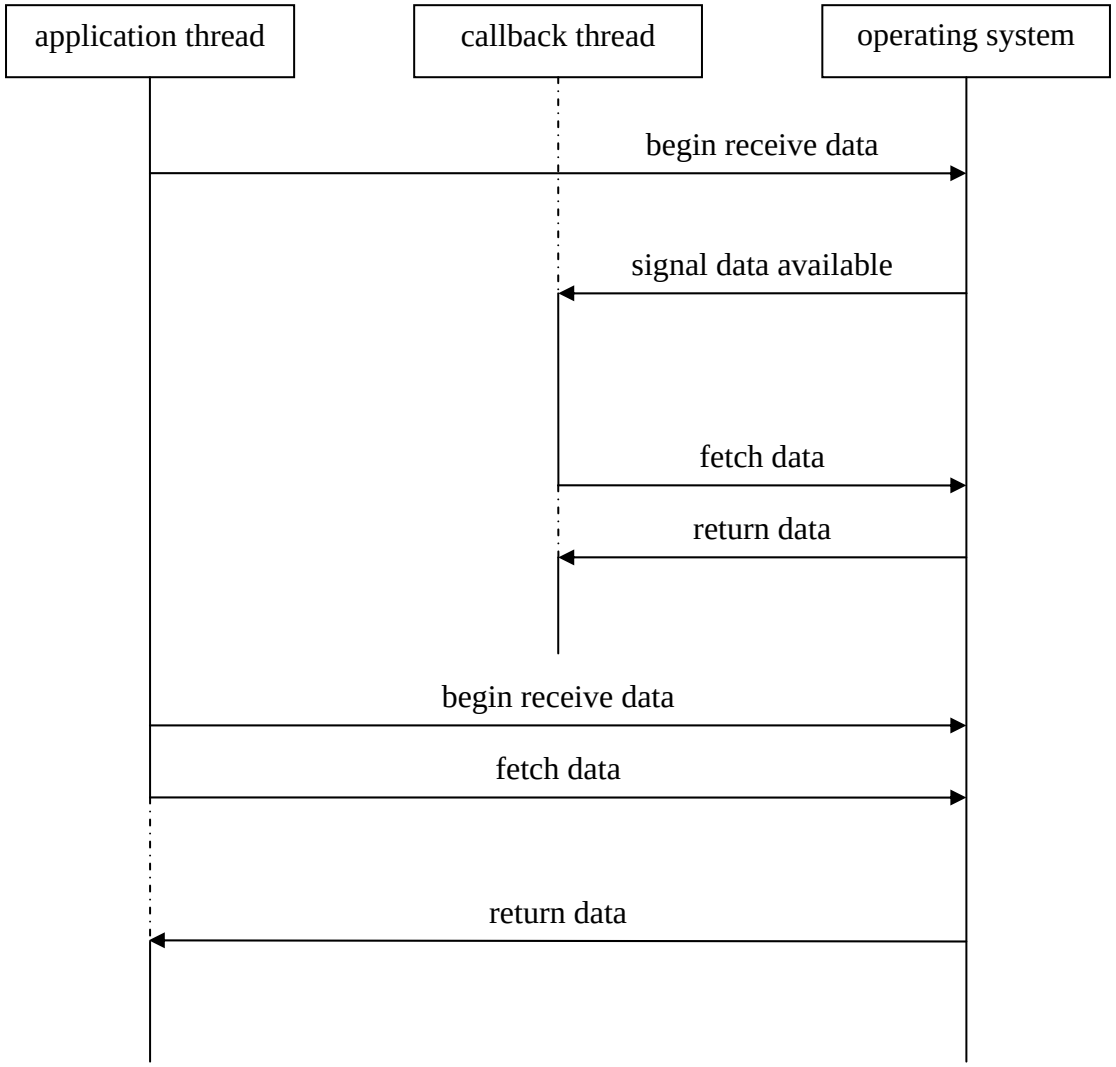

Figure 2.9: An asynchronous data transfer. Dashed lines indicate suspended thread state. In the first case (above), a callback function is registered and is called upon completion of the operation on a separate thread. In the second case (below), no callback is hooked and the initiator blocks until the operation completes.

from a document and the queue was empty when the URLs were appended. If 50 threads are retrieving documents in parallel, 45 are likely to be directed to a single host (see Section 2.7). If the document is the product of a computation-intensive operation, the server may easily run out of resources. In this case, error messages will be returned instead of proper replies. This phenomenon gives rise to two demands:

- A crawler should maintain only a limited number of simultaneous connections to a single web server.
- The frequency of queries should be low.

Many techniques exist to balance the number of concurrent connections to a single server. URL reordering prioritizes (or shuffles) URLs so that they are distributed evenly in time as the URL list is processed in order. On the other hand, a crawler could keep track of the number of concurrent connections to a server and delay requests if they would overload a given server. For a breadth-first traversal, this could be thought of as a weak strategy.

2.4 Content parsing

Filters Due to the inhomogeneity of the Internet, crawlers not only have to respect different standards and versions of these standards but also have to be prepared to handle various character sets, languages and, most notably, document formats. Once downloaded, parsing documents is only possible provided that each document is converted to a uniform format a hyperlink extractor or a search engine indexer can comprehend. Components that perform this conversion are commonly known as *filters*. Most often, though not exclusively, filters convert binary file formats to text streams, which are in turn read by a tokenizer or a hyperlink extractor component to split the document into a series of keywords or find URLs in the document.

IFilter [4] components, either commercial or available for free, facilitate conversion of popular document formats through a well-defined interface. Document formats for which IFilters are available include RTF, PDF, CHM, PS, HTML, XML files and even SQL Server [6] tables. In particular, Microsoft Indexing Service is based on the IFilter technology.

Resolving relative URLs Traversal strategies discussed in Section 2.1 rely on *absolute* URL*s* to traverse the web. A URL is absolute if it identifies a web resource by itself without any external context information. On the contrary, downloaded documents often contain *relative* hyperlinks, which should be interpreted with respect to the document in which they occur. For example, the URL `http://en.wikipedia.org/wiki/Markov_chain` is

absolute but the hyperlink `/wiki/Mathematics` found in the very document identified by the first URL is relative. In particular, it should be resolved to an absolute hyperlink by adding the host address `http://en.wikipedia.org`. Additionally, some URLs are not host-relative but document-relative. The URL `Mathematics` is a document-relative hyperlink and is resolved into `http://en.wikipedia.org/wiki/Mathematics` if found on the web page `http://en.wikipedia.org/wiki/Markov_chain`.

2.5 The hyperlink filter

During web traversal, a crawler must combat various hazards, known as *crawler traps*. A common variant of crawler traps are infinite structures, which are often the result of dynamically generated pages. For instance, in the case of a web calendar the user may navigate to the previous and the next month by means of hyperlink. If a crawler discovers such a calendar, it may follow hyperlinks without limit and download dozens of pages without any relevance. [29]

No general preemptive solution to these cases exists. Dynamically generated content is often created based on the query string appended at the end of the URL. For instance, a specific month for a web calendar may be referenced as:

```
http://example.com/calendar.php?year=2006&month=6
```

The crawler may avoid downloading a multitude of documents with no relevance if it refuses to follow hyperlinks of the above form or follows them with a depth limit. Nonetheless, not all infinite structures are filtered by discarding hyperlinks with query strings. URL rewriting, available to popular web servers including Apache [2] and Microsoft IIS [5], allows hyperlinks to be transformed on the fly. For instance, the "previous month" or "next month" hyperlinks of the fictitious web calendar may appear as:

```
http://example.com/calendar/2006/6
```

Once the HTTP request to the page is received by the web server, the URL is transformed back into the query string version. If incautious, the crawler will traverse the unlimited structure even if it automatically discards hyperlinks with query strings. Only a per-host page count can stop the crawler from infinitely traversing the site albeit a large number of irrelevant pages will have already been gathered by that time.

2.6 Traversal constraints

Not every page is meaningful from the perspective of a crawler. Clearly, administrative pages, data submission forms and search pages are pointless to be visited by a crawler as they contain no useful information a search engine should index. Pages of the English Wikipedia project [9] that allow editing articles are excellent examples. Even though each Wikipedia article features an `edit` hyperlink at the bottom of the article, a crawler should not follow these URLs. Fortunately, these URLs all point to pages in a dedicated subdirectory and a standard protocol, called *Robots Exclusion Protocol* [24] is available to inform crawlers that certain subdirectories should be exempt from being traversed, indexed or archived.

The Robots Exclusion Protocol describes the format of a special file named `robots.txt`, located in the root directory of the host. This is a text file that contains instructions for each type of crawler what documents it should not attempt to download. For instance, Wikipedia forbids UbiCrawler [13] to visit any pages on the Wikipedia site:

```
User-agent: UbiCrawler
Disallow: /
```

In addition, Wikipedia also prohibits crawlers to visit pages in the `/w/` subdirectory relative to the host root. This directory contains the edit pages that allow users to modify existing Wikipedia articles or create new ones. `*` is a wildcard character that maps to the name of any crawler.

```
User-agent: *
Disallow: /w/
```

The Robots Exclusion Protocol is not constrained to host-scoped rules specified in the `robots.txt` file. Crawler instructions can be placed within the HTML documents themselves in the document header (`head`) as meta-information (`meta` elements). Graceful crawlers take this meta-information into account before extracting hyperlinks from a page (for the instruction `no-follow`) or saving the page for future use (for `no-archive`).

In addition to host-specific constraints, other larger-scale constraints may also have to be obeyed. For instance, a crawler might be restricted to a certain primary domains (such as `.hu`) or to sites in a certain language. Also, some sites may have to be manually excluded from the crawl. Either case, a crawler should be configurable to avoid undesirable domain or host visits. For instance, a language-specific crawl could use an N-gram method to discover the language of the page [31] and discard the page without following hyperlinks if it is not written in the target language.

2.7 URL locality

Cho and Molina conducted experiments as to how URL distribution between independent crawler processes affects the volume of inter-process communication. They defined communication overhead as the ratio of exchanged URLs to the number of downloaded URLs. Having differentiated between URL-based hashing and site-based hashing, they inspected how communication overhead increases with the number of processes. In URL hashing, the one-way function was calculated for each URL and forwarded to the responsible process based on the function value. In other words, URLs were assigned to processes. In the other scheme, the hash function was calculated for the host name part rather than the whole URL and the URL was assigned accordingly. Cho and Molina found that while URL hashing leads to an unacceptably sharp increase in communication overhead, site hashing shows a decreasing rate of change with the increase in the number of processes and has an overhead ratio of less than 1 even for 64 independent processes. [18]

In addition, they inspected the ratio of inter-site and intra-site URLs located on a host based on a 40-million-page sample of the web. They found that only a mere 10% of hyperlinks references pages external to the given host. While this by itself means a considerable decrease in the number of exchanged URLs in a site hashing scheme, Cho and Molina found that batch communication does not significantly curtail the quality of the collected pages for large datasets while it further reduces overhead. They defined quality as:

$$\frac{|A_N| \cap |P_N|}{|P_N|} \tag{2.1}$$

Here, A_N represents the set of pages downloaded by a parallel crawler and P_N the set of pages retrieved by an *oracle crawler*. The oracle crawler is an omniscient agent that downloads only those pages that actually satisfy a given importance metric at a given instant. For instance, the PageRank algorithm defines relevance with the recursive formula ($PR(A)$ denotes the PageRank of page A, $C(A)$ the number of distinct outgoing links from A, $R(A)$ is the set of pages that reference A and d is a constant) [15]:

$$PR(A) = (1-d) + d * \sum_{t \in R(A)} \frac{PR(t)}{C(t)} \tag{2.2}$$

Clearly, an actual crawler without an a priori knowledge of importance metrics cannot download all relevant pages. In addition, parallel crawlers perform more poorly than centralized crawlers because even information available to them is scattered and not wholly known to any agent. Surprisingly, quality can significantly increase even if only a few exchanges take place during the entire crawl. In a scenario of 8 million pages, 64 agents and

backreference count as the importance metric (calculated based on 40 million downloaded pages), quality increased from 0.3 to 0.4 with only 10 exchanges during the entire crawl. [18]

Chapter 3

Related work

Despite their wide-spread usage in general or focused search engines, details of high-speed web crawlers that allow reproducibility can scarcely be found in literature. In the following sections, three notable exceptions to this rule will be presented in detail. *Mercator* is a high-speed, centralized crawler architecture with a high level of configurability. *PolyBot* is a distributed crawler decomposed into a replaceable crawling application responsible for traversal strategy and a more general crawling system for performing efficient document retrieval. *UbiCrawler* is a fully distributed fault-tolerant crawler with no central coordinator.

3.1 The Mercator crawler

Mercator [23] is a scalable, extensible web crawler implemented in Java. The system has a pluggable component architecture: the various steps in the basic crawler algorithm in Figure 1.1 are realized by one or more components. Mercator is primarily centralized: it is a multi-threaded application, in which each individual *worker thread* executes the crawler algorithm separately. The scalability of the system is catered for by three factors:

- multiple instances of a given component, possibly one for each worker thread, in bottlenecks,
- a bounded memory use even for large data and
- an efficient access of disk-resident structures to store data that does not fit into memory.

The components of Mercator are derived classes implementing well-defined interfaces. The system is comprised of the following standard building blocks (Figure 3.1):

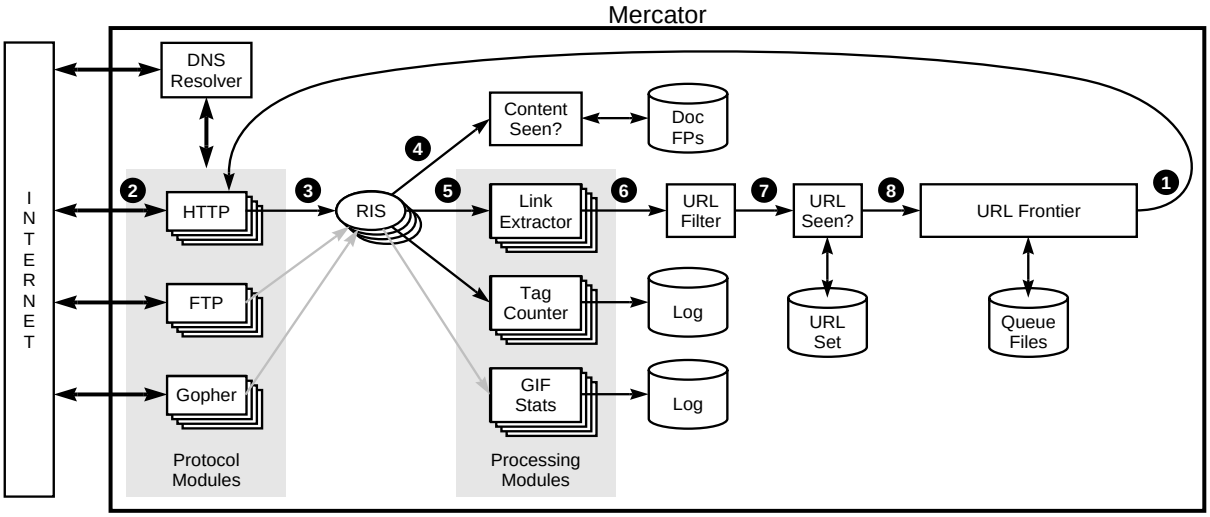

Figure 3.1: The architecture of the Mercator crawler.

- The URL *frontier component* stores and schedules URLs to download. It encapsulates both the traversal strategy (breadth-first search in the reference implementation) and the download policy (web servers currently contacted by a worker thread are not polled simultaneously on other threads). Scheduling is implemented by means of per-worker-thread FIFO queues. When a URL is added to the URL frontier, its host name is canonicalized and the URL itself is appended to one of the queues based on the canonical name. URLs with the same host are appended to the same queue, which guarantees that only one thread is accessing a single host at a time.

- *Protocol modules* are Mercator's components that actually perform document retrieval. Protocol modules include implementations for HTTP, FTP and gopher protocols. A separate instance of each protocol module is associated with every worker thread to avoid cross-thread synchronization. In addition, protocol modules implement the robot exclusion protocol. [24]

- A *rewind input stream* is an I/O abstraction used by Mercator to provide transparent access to downloaded data. A rewind input stream stores backing data either in memory for small amounts or on disk for larger amounts. It allows data to be re-read by the various *content-processing modules*. Each thread possesses a single rewind input stream instance, which is re-used in each crawler algorithm cycle.

- After downloading a document, a *content-seen test* is performed to see if the document has previously been processed before. The content-seen test computes the document fingerprint according to Rabin's fingerprint algorithm and performs an existence check in a memory cache or, if not found, in the disk-resident fingerprint

file. The file is organized so that the check can be performed with a minimum number of disk block reads.

- Content-processing modules read and parse data from the rewind input stream. Similarly to the frontier component and protocol modules, they are pluggable but in contrast to those components, many content-processing modules may be associated with a given document. The most notable content-processing module is the *link extractor module*, which parses document for outbound URLs.

- URL *filters* provide a mechanism to sort out URLs that are not intended to be downloaded prior to their being added to the URL frontier. Mercator URL filters can be combined in conjunction or disjunction, may be negated and their set extended with user-defined filters.

- The URL-seen test checks if a given URL has been encountered before. Similarly to the content-seen test, the URL existence check is performed by means of a one-way function. The one-way function is computed separately for the host part and the entire URL, which prevents locality information from being lost. Locality information allows an efficient hierarchical organization of a disk-resident URL store. Combined with least-recently used caching in memory, this caters for a sharp increase in efficiency and speed.

Evaluation Mercator is a highly-configurable crawler with pluggable components that allow customization to various scenarios. By implementing intricate caching mechanisms, it minimizes random disk accesses. It features a clear application and threading model, which lends itself well to future extension.

While effectively handles large sets of data and provides flexibility by means of pluggable components, Mercator retains a centralized architecture, which limits its applicability for easy scaling. Even if multiple Mercator instances are run on several machines, they should share disk-resident data structures. Despite the disk-seek minimizing techniques of Mercator, fast disks are required so that the application should scale.

In addition, Mercator worker threads are strongly associated with the crawler algorithm rather than data processing. While helps easy authoring of extensions, this decreases the flexibility of the architecture. Distributing each step of the algorithm across several machines is non-trivial as it requires some form of cross-machine synchronization.

Apart from host name locality, further locality information present in URLs is not exploited. For instance, several URL host names may correspond to the same web server, Mercator does not seem to take this into consideration. As worker threads are assigned

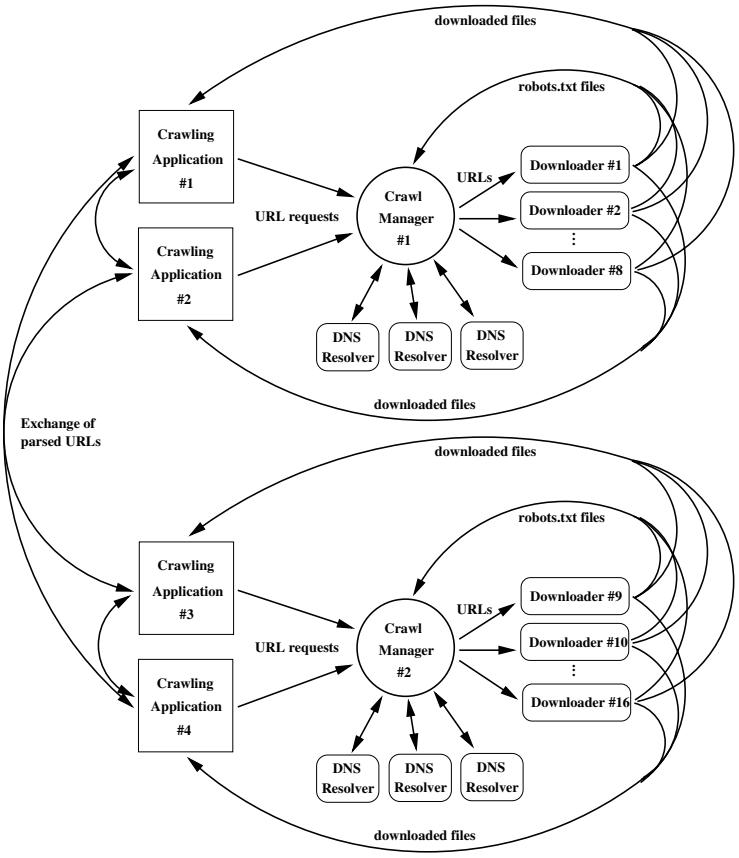

Figure 3.2: The architecture of the PolyBot crawler.

jobs from queues to which URLs are appended based on host name, one cannot control exactly how many threads access a given web server.

3.2 The PolyBot crawler

PolyBot [28] is a scalable, distributed web crawling architecture implemented in C++ and Python. It is partitioned into two distinct parts: *crawling application* and *crawling system*. The customizable crawling application encapsulates the traversal strategy and forwards URLs to download to the crawling system. For instance, a crawling application realizing breadth-first traversal keeps track of URLs already visited and forwards only new URLs to the crawling system to download. The crawling system, in contrast, is a more general part, which is responsible for tasks that are the same regardless of the traversal method used, such as document retrieval, robot exclusion or DNS resolution. The crawling system can be further decomposed into individual components. (Figure 3.2)

The crawl manager is the primary component of the crawling system that directly receives a list of URLs to download from the crawling application. First, the crawl manager

extracts host names from the URLs and forwards them to DNS *resolvers* to obtain IP addresses. Second, it downloads `robots.txt` files from each server unless it has a recent copy of the file. URLs that do not satisfy robot exclusion constraints are removed from the URL list, while the remaining URLs are sent to one or more *downloaders*.

Additionally, the crawl manager is responsible for scheduling document retrieval from various hosts. The manager maintains *ready host* and *waiting host* data structures. The waiting host queue is a data structure sorted by time left before next contact in descending order. Hosts for which the required waiting time has passed are replaced into the ready host queue. URLs belonging to ready hosts are forwarded to the downloaders in batches. Both ready and waiting queues are implemented in PolyBot as B-trees.

The components of the crawling system can be spread across multiple machines. Components communicate by means of either TCP/IP for small control messages or *Network File System* (NFS) for exchange of large amount of data. NFS [27] is strongly though not exclusively associated with UNIX platforms and allows a computer to access a file over the network as if it were available on a local disk. Each PolyBot component receives incoming data as an NFS file pointer, which is read and parsed by the component. Similarly, outbound data is also written to a file accessible via NFS. For instance, downloader components receive an NFS directory with the list of URLs to download and are to place data files they retrieve into this directory.

The other major part of the design, namely, the crawling application, parses files generated by downloads looking for URLs. Parsing is performed with the help of Perl-Compatible Regular Expressions (PCRE). [21] Extracted URLs are matched against an URL-seen structure. This structure is implemented as a red-black tree [30] and caters for bulk lookup and insertion. Newly encountered URLs are inserted into the structure. Once the structure grows beyond a certain size, it is merged with a disk-resident structure. As a result, the memory-resident structure will only contain URLs that should be forwarded to the crawling system to be retrieved. In a typical scenario, pages contain about 8 hyperlinks each. Hence, the number of URLs to download grows in an exponential fashion so that the crawling system has enough work even if hyperlinks from newly retrieved documents will be added to the download list much later than extracted due to the bulk operation.

As the crawling system, the crawling application can be replicated over several machines. However, unlike crawling systems, which are fairly independent, crawling applications should co-operate to partition the web into disjoint sets. This is most easily implemented by means of a hash function, which maps URLs to machines. If a crawling application encounters a URL it is not responsible for, it forwards it to the appropriate application.

Evaluation PolyBot is a distributed web crawler designed to run on a local network of workstations. The system is built up of independent components each responsible for a well-defined task, while NFS provides a way to share data between these components. The architecture scales computation-wise by adding extra components. Some degree of configurability is provided by replacing component implementations.

Nevertheless, while exposes some degree of configurability, this is not always sufficient because configurability is possible through the replacement of relatively large blocks. In particular, the crawling application is not partitioned into smaller, pluggable components though some functional generalization or decomposition would be possible in the case of application as in the case of crawling system. In addition, the architecture seems not to exploit the locality present in URLs, namely, most URLs on a page point to the same site, host or secondary domain.

NFS, while offers good performance when run in a local area network, incurs a larger overhead when used across a wide area network. PolyBot data transfers can be rather significant in terms of size, which constrains the spatial distribution the system permits. Thus, network load dispersion by placing crawlers near the subset of the web they are expected to traverse is limited. PolyBot can only exploit geographical locality to a small degree.

3.3 UbiCrawler

UbiCrawler [13] is a fault-tolerant fully distributed web crawler implemented in Java. It consists of a set of independent, identically programmed *agents* that co-operate without a central coordinator. Agents communicate via Remote Method Invocation (RMI). Each agent is responsible for a set of hosts, that is determined by a one-way function.

As UbiCrawler has no distinctive coordinator, each agent should be able to locally determine where a URL the agent is not responsible for should be forwarded. This is performed by means of consistent hashing. Unlike regular hashing, consistent hashing allows dynamic addition or removal of buckets (or in this particular scenario, agents) without invalidating many of existing (host) mappings. More precisely, in the case of regular hashing, the bucket to place a new item into is determined by computing the hash value over the set of buckets. If a new bucket is added, not only does the set grow but previous mappings will not hold over the extended set. To maintain consistence, all items have to be re-hashed. In consistent hashing, this is overcome by defining hashing over a unit-length circle. Each bucket is replicated k times and replicas are placed randomly over the unit-length circle. When determining into which bucket a new item should be placed, a hash value over the unit-length circle is computed. The item is placed into the

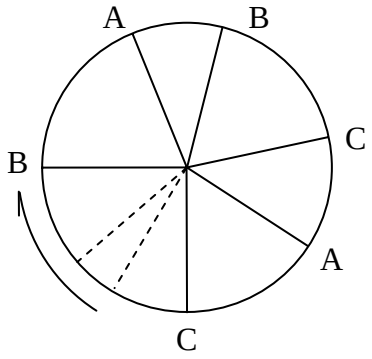

Figure 3.3: Consistent hashing with $n = 3$ and $k = 2$. Both of the two new items (indicated with dashed lines) map to bucket B as it is nearest in a clockwise direction. In case B is removed, only the arcs between (1) southern C and western B and (2) northern A and northern B will have to be re-mapped to A and C, respectively. The re-map cost decreases as n and k increase.

bucket to whose replica the hash value is nearest in a clockwise direction. (Figure 3.3)

In addition to the property of consistent hashing that allows dynamic addition and removal of agents, UbiCrawler agents should have an identical picture of the distribution of replicas over the unit-length circle. Hence, replicas are placed as given by a pseudo-random number generator seeded with the unique identifier of the newly added agent. As the unique identifier is announced to all previously existing agents, each can compute the location of the new replicas. The proportion of URLs that now reside on the wrong agent remains relatively low.

Identifier-seeded consistent hashing in UbiCrawler guarantees an even distribution of hosts over the set of agents, even though the number of pages per host may vary. Additionally, UbiCrawler agents have a property called *capacity* that is a function of agent hardware performance and network bandwidth. When replicas are created for an agent, the number of replicas placed along the unit circle is proportional to the capacity of the agent.

The traversal of the web is performed in a local breadth-first manner. When an agent receives a URL associated with a host not seen before, it traverses all pages belonging to the host breadth-first. The traversal is carried out on a dedicated thread. URLs to foreign hosts are forwarded to the appropriate agent. As all pages related to a host are visited in a single batch, UbiCrawler implements no special caching for results of DNS resolution or robot exclusion files.

Evaluation UbiCrawler has a fully scalable architecture with the individual agents being the unit of replication. While the degree of distribution is relatively coarse-grained,

agents are kept simple and can hence run a single machine each without difficulty. As the architecture is horizontal and lacks any hierarchical aspect, the system comprised of several agents is very flexible and fault-tolerant.

Nonetheless, the simplicity of agents can become a drawback in some usage scenarios. Agents use no special data storage or caching mechanisms but sacrifice effectiveness for the sake of low resource use. In particular, the host-specific breadth-first traversal can be suboptimal in many cases: not all pages of a relevant host are relevant in a global sense, which UbiCrawler assumes when traversing all pages of a host in succession.

Tasks requiring intensive computation that could be solved by further distribution are hard to incorporate into the model. For instance, an image processing job cannot be handled by a separate agent. In fact, UbiCrawler exposes no high-degree configurability.

Chapter 4

The component architecture

As introduced in Chapter 1 and as seen in related work in Chapter 3, despite the apparently simple basic algorithm of any crawling system, large demands in terms of data storage and computing capacity necessitate either very efficient usage of space and processor time or, as recent research shows, a decomposition of a crawling system across multiple machines. On the other hand, easy reconfigurability encourages the use of pluggable components and on-the-fly parameter modification. Nonetheless, no description of an efficient crawler that possesses the benefits of both scalability and easy configuration has been published.

In order to tackle the challenge, the authors propose a distributed architecture. Usually, in spite of the scalability a well-designed distributed crawler provides, the complexity of the system increases sharply. Connection management, data transfer and error recovery become critical issues. Nevertheless, communication overhead is external to the functional aspect of the crawling application. Our proposal is that functionality can be wrapped into *components* that perform some transformation on their input and produce some output. For instance, a parser component receives documents and produces hyperlinks contained in the document. This encapsulation is valid even if components rely on shared data sources. For example, a downloader component receives URLs to download, contacts a DNS service to resolve the host name into an IP address and fetches the document by generating an HTTP request to the obtained address, which constitutes the output of the component for each URL.[1] In other words, the crawler is comprised of loosely-coupled functional units between which data flow can automatically be marshaled.

For the reasons discussed above, it seems straightforward to design a framework that shields the component developer from the intricacy of the distributed architecture. The framework will be responsible for component initialization and configuration based on XML documents, component lifecycle management and (possibly on-demand) establish-

[1] In the terminology of our system, the component has utilized the services of a *provider*.

ment and management of remote connections. From the perspective of components, the architecture is thus simplified into an input, an output and any number of service or data provider interfaces. The actual complexity of the design will remain hidden from view.

4.1 Terminology

The distributed component architecture is realized over a set of computers (or *machines*) connected by a network. Each computer is running one or more framework *processes*. These processes are isolated and always act as if they were distributed across multiple machines even if they are co-located on the same computer. The network protocol used for remote communication is treated as reliable, i.e. no loss of data due to network congestion or failures is assumed. The standard TCP/IP protocol, for instance, satisfies this constraint. Within each framework process, one or more *threads* may be executing. Threads are independent in terms of processing time they get but may share data structures visible to each.

The building blocks (or *units*) of the proposed component architecture rely heavily on object-oriented design. They see one another through interfaces and are realized as classes implementing these interfaces. The actual objects behind the interfaces are instantiated by the framework, which sets the appropriate references to each newly instantiated object behind an interface. This process is called *binding*. For instance, six downloader components may use a single DNS resolver behind an `IHostResolver` interface to fetch IP addresses for a host name. We say that the downloaders *bind* to the resolver through the interface or the resolver *is bound* to six downloaders.

4.2 General structure

Our architecture closely resembles the *pipes and filters* pattern common to Unix-flavor operating systems. In this model, *filters* transform data, while *pipes* act as a buffer and marshal data from one filter to the other as it is being produced. Nevertheless, our architecture differs in various aspects from the Unix pattern. In the Unix version, data that flows between filters is unstructured, that is, it is a stream of characters, which is intended to model the common scenario when the output of one program is used as an input of the other. In our architecture, the counterparts of pipes, called *connectors*, are FIFO queues of structured data. They can also span across machines and cater for multi-producer and multi-consumer scenarios. The counterparts of filters, called *components* in our system, are also more relaxed in terms of how they accept input. In addition, our system is a pull rather than a push model: components have fair influence in when they

consume input or generate output. In the sections that follow, we give further details of our system.

4.3 Classification of building blocks

Our architecture distinguishes three distinct types of building blocks. *Components* and *providers* share some functionality and they encapsulate functional behavior. *Connectors* are responsible for data exchange and buffering.

- *Components* are the primary constituents of the architecture. Components generally consume, transform and produce data. They encapsulate user-defined functionality wrapped in an automatically invoked method, which performs the actual computation based on input and generates respective output.

  Components are data transforming units. For instance, a unit that verifies a URL against robot exclusion rules in effect for the given host and discards URLs that do not match constraints can be classified as a component.

- *Providers* offer access to external or shared data sources. Providers may be bound to either components or other providers.[2] As the methods of providers may be invoked concurrently by multiple components or other providers, implementors should synchronize access to data structures providers use.

  Providers are either a means of marshaling external data, of utilizing machine-specific resources or of cross-component synchronization. For instance, a unit that executes a stored procedure through a database connection from a pool is a data provider. A unit that downloads and caches robot exclusion files is service provider.

- *Connectors* are the glue between components and drive data flow in the distributed application. They are usually finite-capacity FIFO queues, which components may place items into or remote items from. Connectors can be either local or remote. A local connector stores references of data items and glues two components in the same process. A remote connector transmits data through a network and hence *serializes* data. During serialization, items are converted into a (machine-independent) byte stream representation, based on which data can be reconstructed once it has been sent through the network.

Connectors, components and providers form a directed graph. If we remove provider nodes from the graph, the resultant graph will be bipartite. Figure 4.1 shows a sample system with connectors, components and providers.

[2] Circular referencing, of course, is not permitted.

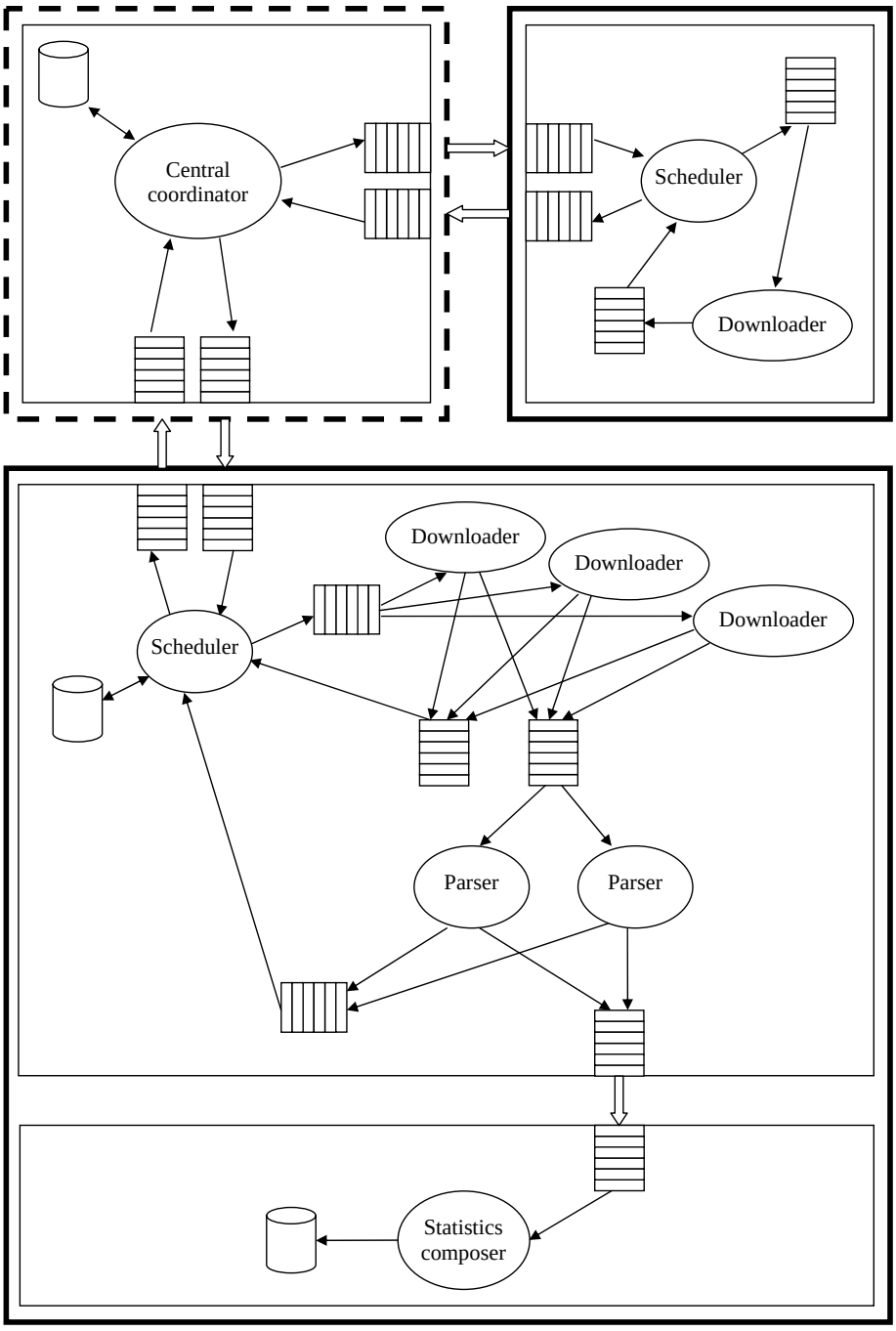

Figure 4.1: A sample component architecture system. Components are depicted as ellipses, connectors as subdivided rectangles and data providers as cylinders. Process boundaries are shown by thin container rectangles. The boundary of the server is indicated by a thick dashed line, the boundary of clients by thick continuous line. Arrows with full head show pass-by-reference, arrows with empty head serialized network data transmission.

4.4 Connectors

As briefly mentioned in Section 4.3, connectors are temporary stores of items. Connectors are *strongly typed,* i.e. they are parameterized with a reference or value type upon startup and accept only elements of this type.[3] In addition, as the fact whether a connector is local or remote is transparent to the components the connector is bound to, connector types must also be serializable. Serializability allows data to be transmitted over the network should the degree of distribution be increased in the system by scattering components across machines.[4]

Though our system includes a specific connector implementation that realizes a finite-capacity FIFO queue that suits most scenarios, user-defined connectors may also be implemented. For instance, a user-defined connector could give priority to certain types of data items. This extensibility is made possible through interfaces a connector should implement.

- `IInputConnector<T>` is the interface through which consumer components see the connector. It provides methods `Consume()`, `ConsumeAsynchronously()` and `BulkConsume()`. `Consume()` retrieves a single item from the connector synchronously, i.e. the caller thread is suspended if no items are available. On the contrary, `ConsumeAsynchronously()` allows the caller thread to resume execution and invokes a registered callback method once an item is available. `BulkConsume()` supports consumption of multiple items in one batch.

- `IOutputConnector<T>` is the producer counterpart of `IInputConnector<T>`. It provides similar methods with functionality adapted to the producer scenario.

Connectors may be accessed by multiple components simultaneously. It is thus imperative that implementors provide the necessary mutual exclusion mechanisms to avoid data corruption.

As previously noted, connectors may either be local or remote. Local connectors pass references of data items and perform data copying only for value types. Remote connectors involve a network connection. However, remote connectors are an abstraction and not an actual connector type. They comprise of (a) a local connection and a sender component that binds to the connection on the local side; and (b) a receiver component

[3] For reference types, due to object-oriented behavior, connectors accept inheritors (i.e. specializations) of their connector type.

[4] Serialization is currently performed by means of .NET's built-in binary serializer. For greater flexibility, this could be replaced by XML-based data exchange or a platform-independent communication protocol such as GIOP.

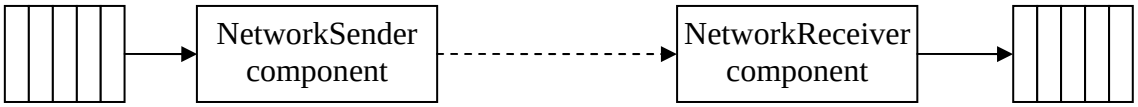

Figure 4.2: The internal structure of a uni-directional remote connector. The continuous line indicates pass by reference, the dashed line corresponds to serialized data transmission through the network.

and a local connection bound to the receiver on the remote side.[5] (Figure 4.2) In order to save bandwidth, data exchange between remote parties is usually performed in batch.

Behavior Connectors offer *partially asynchronous* communication between components. By being transparent temporary item stores, they represent finite buffers that transmit messages. They are asynchronous because once an item is fed into a connector, no indication is ever sent to the producer component when the transmission was successful.[6] On the other hand, they provide some form of synchronization because if the connector becomes full, new items are not accepted. (Note that items are fed to the connector by means of a synchronous method.) This means that the producer component is suspended until free slots are available. Thus, though basically asynchronous, connector data transfer shows some synchronization properties.

Access mechanism Connectors are implemented through a three-gate lock mechanism. First, a component that wishes to produce items must seize a producer *mutex*. The mutex prevents other components from feeding items to the connector before the initiator component has finished, which allows contiguous blocks of items to be produced by a single component without interleaving. Second, the component must seize a *slot available semaphore* as many times as the number of items it wishes to produce. This ensures that the number of free slots the component intends to occupy are actually at its disposal. Third, the component thread enters a *critical section* and actually adds the items one by one to the connector. The critical section prevents consumers from simultaneously accessing the connector while the items are being placed, which could otherwise lead to data corruption. Whenever it cannot satisfy the criteria to enter a "gate," the component thread is suspended.

After an item has been added to the connector and the critical section has been left, an *item available semaphore* is released, which allows consumers waiting for new items to resume execution. Finally, the producer mutex is released so that other producers can

[5] If data transmission is bidirectional for the connector, which is seldom the case, both sides should have sender and receiver components.

[6] The architecture does not lose items during data exchange.

attempt feeding items to the connector.

The consumer side of the access mechanism resembles the producer side with consumer mutex instead of producer mutex and the role of the slot available semaphore and the item available semaphore replaced.

4.5 Components

Although components realize a large variety of different functionality, the framework implements basic operations that allow the component to bind to connectors, to be configured via XML files or to be started or suspended upon need. Hence, in the framework we propose, all components derive from `GenericComponent`, which is the root of the component hierarchy. `GenericComponent` exposes the following methods:

- `Initialize()` is invoked immediately after all providers and connectors the component references have been bound and the component has been configured. The `Initialize()` method is executed synchronously by the configuration thread. (See Section 4.9 for defined thread types in the framework.)

- The framework initiates the component by the `Start()` method. The `Start()` method is assumed to return, possibly by initiating an asynchronous but not necessarily finite operation.

  Most components realize the *single-threaded component model*. Here, a dedicated thread is associated with the component. Data is extracted from input connectors, processed, and the result of the transformation is fed into output connector queues on this dedicated thread. This eases component development as no mutual exclusion devices are required on data structures. The framework provides the `GenericThreadedComponent` class to develop single-threaded components.

- The `Suspend()` method instructs the component to temporarily suspend operation. This method is intended to reverse the effect of `Start()`.

- The `BindConnector()` family of methods aid binding connectors to the component. They interpret component annotation (see Section 4.5.1) and perform type validation. These methods are invoked by the framework to configure the component.

- Enumerator methods return input, output or all connectors bound to the component. They are used by component types that perform connector bindings based on component annotation. (See Section 4.5.1.)

- `Dispose()` releases all resources claimed by the component during initialization.

For components realizing the single-threaded model, additional semantics is defined. The `Execute()` method is automatically invoked upon startup by the `Start()` method on a separate thread and is expected to continually execute until an exit condition is satisfied. The exit condition is usually associated with connectors bound to the component, e.g. all elements in the connector queues have been consumed.

4.5.1 Component annotation

In modern object-oriented programming languages, annotations are a means of associating meta-information with classes. Our architecture uses annotation services (called attributes in .NET terminology) to describe component to connector bindings whenever the binding is not obvious. For instance, for a component that consumes from three input queues and produces to two output queues it is essential to know the exact types of items. Also, it is important to distinguish between two connectors both of which are sources of the same type of objects yet serve different purposes.

Our architecture has three types of component class attributes. Each attribute may appear multiple times.

- `InputConsumer` and `OutputProducer` attributes specify the type of items consumed from or produced to a given connector. The connector is uniquely identified by means of a *role*. During the initialization phase, the XML configuration file is read and parsed. When setting up components, binding is performed with the help of a connector identifier and the role. The `BindConnector()` family of methods is passed the connector instance and its role. These methods may then set the appropriate member variables within the component instance.

- The `DataConsumer` attribute resembles the `InputConsumer` and `OutputProducer` attributes but it associates a data provider rather than a connector with the component. Similarly, it takes a role and a type parameter. However, here the type parameter identifies the type of the expected provider class or interface.

4.5.2 Component types

Components are usually not written from scratch by directly inheriting from the component base class. One can easily identify certain consumer and producer patterns and generalize them. (Figure 4.3) In the following subsections, general component types our system supports will be introduced.[7]

[7] In this context, a phrase such as "a `ComplexFilter` is synchronous" should be understood as "the inheritors of the class `ComplexFilter` are synchronous." In fact, component types are all abstract classes and cannot directly be instantiated.

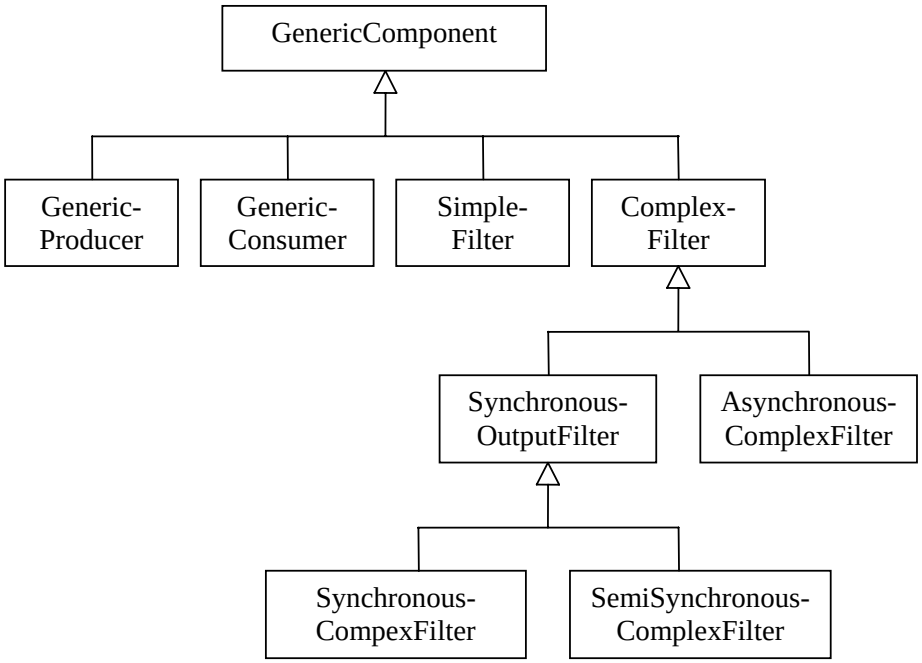

Figure 4.3: Hierarchy of the component architecture. Component implementors should derive from one of the descendant classes rather than directly from the uppermost superclass.

4.5.2.1 GenericConsumer

A `GenericConsumer` is a special component that produces no output yet it consumes input. `GenericConsumer`s are useful in logging and in any other operation that produces only data external to the crawling system. For instance, a crawler that traverses the web for images, creates thumbnails and saves them to disk could use a thumbnail generator component. Clearly, the images the component creates are not fed back into the system.

4.5.2.2 GenericProducer

A `GenericProducer` generates output data but consumes no input. They are handy in generating test input or signals based on state change. In addition to the aforementioned scenarios, *remote connectors* are also realized with `GenericProducer`s, `GenericConsumer`s and regular connectors. The `GenericConsumer` consumes items from a local connector queue and transmits them over the network. On the remote side, a `GenericProducer`, once it has received transmitted data, feeds items back into another connector.

4.5.2.3 SimpleFilter

A `SimpleFilter` is a regular component that consumes items from a single input queue and produces zero or more items for each input item. `SimpleFilter` derivatives should

```
namespace Crawler {
    class SimpleHyperlinkFilter : SimpleFilter<Uri, IPAddress> {
        private List<IPAddress> result = new List<IPAddress>();

        protected override IList<IPAddress> Filter(Uri uri) {
            IPAddress[] ips = Dns.GetHostAddresses(uri.Host);
            result.AddRange(ips);
            return result;
        }
    }
}
```

Figure 4.4: A SimpleFilter. The types the component consumes and produces are encoded into template parameters.

override the `Filter()` method. When data is available in the input queue, the framework invokes the method, which is to return a list of output items. The resultant items are appended to the output queue once free slots are available. The `Filter()` method has the following signature:

```
protected abstract IList<T> Filter(S data);
```

Here, `S` signifies the input data type, while `T` denotes the output data type.

An IP extractor component is a typical example of a simple filter. For each URL it receives, it extracts the host name and queries DNS servers for all IP addresses that correspond to the host name. The addresses are then appended to the output queue. (Figure 4.4)

4.5.2.4 SynchronousComplexFilter

A `SynchronousComplexFilter` is a more powerful `SimpleFilter` variant in the sense that it can consume input from and produce output to multiple queues. However, as the name suggests, the component is synchronous, that is, the `Filter()` method is invoked only if input is available in every input queue. Unlike `SimpleFilter`, the `Filter()` method of a `SynchronousComplexFilter` accepts no input parameters and has no return value. Instead, the framework sets specially annotated properties with the values of input items, invokes the parameterless `Filter()` method, and gets output values from similarly annotated properties.[8] In Figure 4.5, the `SimpleFilter` discussed earlier is rewritten as a `SynchronousComplexFilter`.

[8]In this paper, we do not strictly differentiate between the class members called *fields* (which directly map to values or objects) and *properties* (which actually correspond to possibly a pair of methods) as the .NET documentation [1] does. Instead, we refer to both concepts as *property* and explicitly state if fields are not allowed.

```
namespace Crawler {
    [InputConsumer("inputQueue", typeof(Uri))]
    [OutputProducer("outputQueue", typeof(IPAddress))]
    class ComplexHyperlinkFilter : SynchronousComplexFilter {
        [InputProperty("inputQueue")]
        private Uri uri = null;

        [OutputProperty("outputQueue")]
        private List<IPAddress> addresses = new List<IPAddress>();

        protected override void Filter() {
            addresses.AddRange(Dns.GetHostAddresses(uri.Host));
        }
    }
}
```

Figure 4.5: The `SimpleFilter` in Figure 4.4 rewritten as a `SynchronousComplexFilter`.

In this case, the architecture pattern is more general than in Figure 4.4 as it should be prepared to accept multiple input and output connectors even if not the case in this particular scenario. Hence, types are discovered by inspecting component annotation rather than hard-coding them into positional template parameters. Note that role strings such as `inputQueue` or `outputQueue` are not required in the `SimpleFilter` case. If specifying a single input and output connector in the configuration file, the mapping of these connectors is obvious. In the `SynchronousComplexFilter` case, however, more input and output connectors may exist and the queue item → property assignment is only possible through some form of unique identification.

4.5.2.5 SemiSynchronousComplexFilter

A `SemiSynchronousComplexFilter` is strikingly similar to a `SynchronousComplexFilter` albeit it operates according to different principles. Notably, it supports asynchronous consumption from its input connector queues. In other words, once a new item is available in any of the connectors bound to the component, the corresponding property is set. A `SemiSynchronousComplexFilter` has no `Filter()` method. Instead, inheritors should use `set()` accessors to perform user-defined functionality when the value of an input property is set with a new item from the input connector. Whenever a single input property is set, all output property values or collections are inspected if they are non-null or contain any elements. If so, they are fed synchronously into output queues. (Figure 4.6)

```
namespace Crawler {
    [InputConsumer("inputQueue", typeof(Uri))]
    [OutputProducer("outputQueue", typeof(IPAddress))]
    class SemiSynchronousHyperlinkFilter : SemiSynchronousComplexFilter {
        [InputProperty("inputQueue")]
        private Uri uri {
            set {
                addresses.AddRange(Dns.GetHostAddresses(value.Host));
            }
        }

        [OutputProperty("outputQueue")]
        private List<IPAddress> addresses = new List<IPAddress>();
    }
}
```

Figure 4.6: The `SynchronousComplexFilter` in Figure 4.5 rewritten as a `SemiSynchronousComplexFilter`.

4.5.2.6 AsynchronousComplexFilter

An `AsynchronousComplexFilter`, while resembles a `SynchronousComplexFilter`, is a much more complex component. It is fully asynchronous, i.e. there is no respective order in which input and output properties are get and set. Whenever one or more items are available in any of the input connector queues, the corresponding input properties are set in no particular order. Similarly, whenever free slots are available in any of the output connector queues, output properties are queried.

Figure 4.7 shows an `AsynchronousComplexFilter` implementation. The component binds to two input connectors. The first one accepts URIs, extracts host names and asynchronously resolves names into IP addresses. The `AppendHostAddresses()` method is invoked on a separate thread once IP addresses are ready to be fetched. Note the presence of a lock mechanism. During the invocation of `AppendHostAddresses()`, an IP address may be read from one of the input connectors or the already resolved IP addresses may be written to the output connector. Without the lock mechanism, data structures may easily be corrupted.

4.6 Providers

Providers are the third class of units in the component architecture and encapsulate external data sources (*data providers*) or services (*service providers*) a component relies on. A provider is a regular class that exposes public properties and methods, which can be manipulated by components that bind to the provider. Unlike typical components,

```
namespace Crawler {
    [InputConsumer("inputUriQueue", typeof(Uri))]
    [InputConsumer("inputIPAddressQueue", typeof(IPAddress))]
    [OutputProducer("outputQueue", typeof(IPAddress))]
    class AsynchronousHyperlinkFilter : AsynchronousComplexFilter {
        private List<IPAddress> addresses = new List<IPAddress>();

        [InputProperty("inputUriQueue")]
        private Uri InputUri {
            set {
                Dns.BeginGetHostAddresses(value.Host,
                    AppendHostAddresses, null);
            }
        }

        [InputProperty("inputIPAddressQueue")]
        private IPAddress InputIPAddress {
            set {
                lock (addresses) {
                    addresses.Add(value);
                }
                GetItemAvailableSignal("outputQueue").Set();
            }
        }

        private void AppendHostAddresses(IAsyncResult ar) {
            IPAddress[] ips = Dns.EndGetHostAddresses(ar);
            lock (addresses) {
                addresses.AddRange(ips);
            }
            GetItemAvailableSignal("outputQueue").Set();
        }

        [OutputProperty("outputQueue")]
        private IPAddress[] Addresses {
            get {
                lock (addresses) {
                    IPAddress[] ips = addresses.ToArray();
                    addresses.Clear();
                    return ips;
                }
            }
        }
    }
}
```

Figure 4.7: A sample `AsynchronousComplexFilter`.

providers are not accessed from a dedicated thread. Hence, providers should employ mutual exclusion devices to prevent data corruption.

The typical use of providers is data access. Components usually expect providers that implement a certain interface. For instance, a URL filter component could invoke a URL seen provider to check if a URL has previously been referenced. The provider interface exposes an `IsUrlSeen()` method that performs the test. The actual implementation of the provider queries an underlying database to see if the URL exists. On the other hand, if the actual implementation is replaced by a probabilistic structure, such as a Bloom filter [11], changes are required only in the configuration file. By changing the provider the component binds to, the system will run without recompilation.

4.7 Initial configuration

While configurable dynamically (see Section 4.8), in most situations it is desirable to initialize the set of components and providers that live in the same process in a declarative way. XML files are at the user's disposal for this end. Figure 4.8 shows a simple initial configuration.

This configuration initializes an `MSSQLClientDataProvider` and a `DNSResolver` provider instance, five instances of the `Downloader` and a single instance of the `HtmlParser` component. Additionally, three connectors are brought into existence, two of which are remote connectors consisting of a local connector and a receiver and a sender component, respectively. The network receiver and sender components are initialized behind the scenes. Notably, the network addresses specified for remote connectors instruct receiver and sender components how to marshal network communication.

All providers and components declaratively configured by means of XML files are automatically initialized during startup. For reasons of consistency, the initialization proceeds in the order: providers, connectors, components.

4.8 Configuration interface

Apart from declarative configuration by means of XML files, the architecture allows on-the-fly reconfiguration. Components expose public properties setting which influences the way the component works. For instance, setting a depth limit for a depth-first traversal component curtails the number of levels it schedules for crawling on a given host. In addition, connector and provider bindings can also be changed. As both connectors and providers are seen through interfaces, switching the underlying object does not disturb behavior. For instance, a DNS resolver that uses a binary tree data structure can easily

```
<?xml version="1.0" encoding="utf-8" ?>
<CrawlerSetup>
  <Providers>
    <Provider id="SQLProvider"
      type="Crawler.MSSQLClientDataProvider, CrawlerClient">
      <Properties>
        <Property name="ConnectionString" value="Data Source=.;
          Initial Catalog=crawlerdb;Integrated Security=true;Pooling=false" />
      <Properties>
    </Provider>
    <Provider id="DNSResolver"
      type="Crawler.DNSResolver, CrawlerClient" />
  </Providers>
  <Connectors>
    <RemoteConnector id="HyperlinksToDownload" type="System.Uri" capacity="100"
      bindingAddress="tcp://localhost:1160" />
    <LocalConnector id="DownloadedDocuments" type="System.Uri" capacity="100" />
    <RemoteConnector id="HyperlinksExtracted" type="System.Uri" capacity="100"
      recipientAddress="tcp://localhost:5050"
      identifier="efd335bd-0d47-4ed9-8123-41bce76730e5"
      replyAddress="tcp://localhost:1160">
    </RemoteConnector>
  </Connectors>
  <Components>
    <Component id="Downloader" type="Crawler.Downloader, CrawlerClient"
      instances="5" source="reference:HyperlinksToDownload"
      sink="DownloadedDocuments">
      <Providers>
        <Provider role="DNSResolver" provider="reference:DNSResolver" />
      </Providers>
    </Component>
    <Component id="HyperlinkExtractor" type="Crawler.HtmlParser, CrawlerClient"
      source="reference:DownloadedDocuments"
      sink="reference:HyperlinksExtracted" />
  </Components>
</CrawlerSetup>
```

Figure 4.8: A sample XML configuration file.

be replaced by one that uses a hash table while the system is running.

In order to avoid data corruption, components must be suspended prior to reconfiguration. The `Start()` method of the component serves to launch the component again after properties or bindings have been changed. If new components or providers have been instantiated, the `Initialize()` method is called for each of these before they are bound.

4.9 Threading

In our architecture, three different types of threads are distinguished.

- The *configuration thread* is the main thread of the framework application. Its primary task is to set up connectors, components and providers and initiate component threads when the application starts.[9] Additionally, the configuration thread listens for reconfiguration instructions sent over the network and is responsible for carrying out these actions. It can suspend and resume component threads.

- A *component thread* is associated with each component and manages the component while it is not in suspended state. For synchronous components, the component thread marshals input consumption and output production. For asynchronous components, separate *helper threads* are associated with each binding. Components may also register additional threads for network management, data transfer, etc. Even if multiple extra threads are started by the component, the component thread still constitutes the main thread of the component. Once it terminates, all other threads are aborted and external resources are released, which signals the end of the component lifecycle.

- Worker threads live in a thread pool awaiting for asynchronous jobs to execute. For instance, the DNS resolution in Figure 4.7 uses a worker thread. Worker threads help avoid thread initialization overhead by keeping threads alive even if they have completed the operation assigned to them.

4.10 Inter-component coordination

In order to retain the modular structure of the component architecture, a clear design should consist of independent components possibly utilizing the services of providers. Generally, a crawler adheres to the filter architecture suggested by the framework as data

[9] Component `Initialize()` and `Start()` methods are invoked from the configuration thread.

flows from component to component. For instance, a downloader component produces documents, which are placed into a connector shared by the downloader and a hyperlink extractor. The extractor reads documents from its input connector and produces URLs to its output connector. The two component types are basically independent. However, in some situations, components have to cooperate. Cooperation undoubtedly requires some form of coordination between the participating components. Two natural ways exist for this end:

- *Provider-based co-operation* can be classified into local and remote cases. (Figure 4.9)
  - In the case of *local* provider-based cooperation, two components use the very same provider and may hence share data. For instance, two downloader components may use the same DNS resolver as a provider to get IP addresses for host names. However, this method requires pass-by-reference and hence confines the participating components to run in the same process.
  - On the contrary, components with *remote* provider-based cooperation access the same data source (e.g. a relational database or a network file) via their respective providers (which may coincidentally be identical) and realize data sharing in this manner. For instance, both a global traversal strategy and a host-specific traversal strategy component may operate on the same underlying database.

  In our design, we completely avoid local provider-based cooperation for different component types and opt for its remote version as the latter allows any two components to be separated to two different machines. On the other hand, we use local provider-based cooperation to share data between components of the same type on the same machine. In this latter scenario, the provider encapsulates functionality that should not scale with the number of components. For instance, multiple downloader components may be used to increase download rate but only one DNS resolver is required for the whole process.

- In *connector-based cooperation*, connectors serve as message channels between components. In this model, components are interconnected by connectors so that a component may respond to the data it receives via acknowledgment or other statistical messages. In our architecture, we frequently use connector-based cooperation to send informational messages about the completion of an assigned job. For instance, a downloader component signals downloaded documents to the host-specific selection strategy by emitting informational elements into a connector queue.

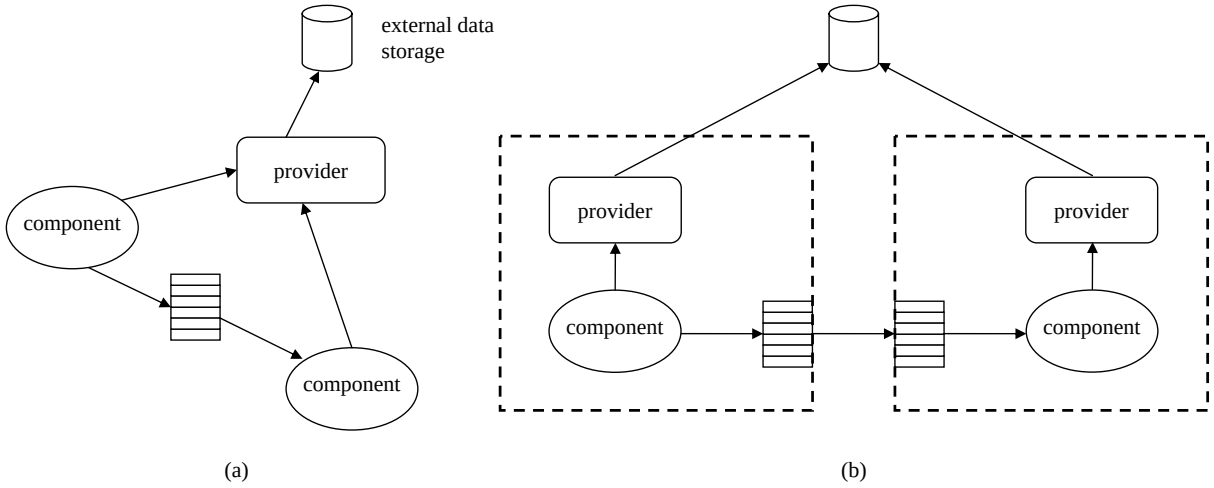

Figure 4.9: Provider-based cooperation. (a) Local cooperation. (b) Remote cooperation. Dashed lines indicate process boundary.

Chapter 5

The crawler application

Despite their complexity in management and overhead in communication, distributed crawlers have major advantages in terms of scalability, dispersion of network load and an overall decrease in network traffic. Computing and data storage capacity increases with the number of crawling agents (scalability), agents can be placed at locations traffic-wise near the web domains they are to process (network load dispersion), thereby requiring less data to be transmitted over the network (network traffic reduction).

The framework introduced in Chapter 4 facilitates the development of a distributed crawler by offering a standard solution to threading, transparent dispersion of components and automated initialization and reconfiguration. Development of the actual crawler can focus on efficient use of CPU, memory, disk and network bandwidth to incur the minimal penalty on the executing system without additional effort to put into data sharing over the network.

5.1 Overview of the system

In the crawler architecture we propose, we distinguish between two participants. A *server* is a central coordinator that assigns hosts and secondary domains to *clients*. A server may take into account the geographical location of the client and various global traversal constraints in effect (e.g. only to traverse the `.hu` domain). Clients are the essence of the architecture which retrieve documents with the appropriate traversal strategy and submit URLs back to the server they are not directly responsible for. Note that neither the server, nor clients are confined to a single or separate machines. In fact, they can all run on the same machine or a client may also be the server. This is made possible by the component architecture that hides remote communication.

Even though we have presented related work [13] that shows strict fault-tolerant properties, we believe that a central coordinator makes the crawler system more manageable

despite its being a single point of failure. Nevertheless, our system is fault-tolerant in the sense that it survives system restarts. Unavailable network connections suspend parties that cannot send data to their recipients. From this perspective, a critical server failure leads only to a temporary out-of-service condition. In fact, clients can continue without delay if they do not depend on external data, which is often the case.

5.2 Client components

Clients are the major unit of replication in our system, each of which connects to a server. Clients are responsible for traversal of hosts or domains and for maintaining related data structures. In particular, each client has its own URL queue and URL-seen structure. (Compare to Figure 1.1) A minimal client configuration and the data flow between its components is seen in Figure 5.1. The roles of the particular components are detailed in the following sections.

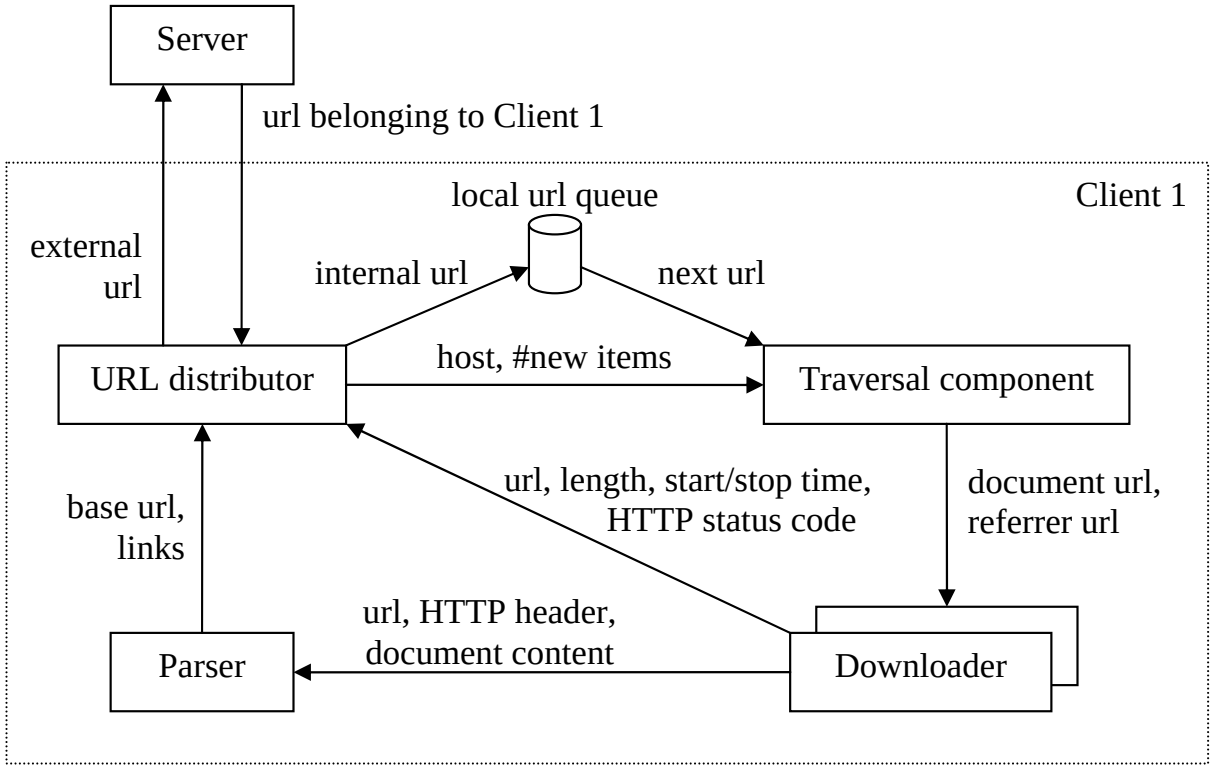

Figure 5.1: A configuration of the system that provides minimum functionality. The rectangles represent the components and the arrows between them indicate the direction of data flow.

5.2.1 Scheduling

URLs to documents awaiting to be downloaded are stored in a URL queue on each client.[1] During scheduling, URLs in the queue are fed to downloaders in a respective order with proper timing. Scheduling is split into two phases: the order of documents is determined by the *traversal component*, while timing is provided by the *load balancer component*.

Scheduling is based on host name. Usually, the number of hosts a client handles is in a range that allows corresponding data structures to reside entirely in memory. On the other hand, when the system is used to traverse millions of web pages, memory use can be spared by differentiating between *active* and *inactive* hosts.

A host is considered inactive if the URL queue contains no element that would be located on the given host. Nonetheless, this does not necessarily indicate that the entire contents of the host has been downloaded. On the contrary, the client may receive further references to the host from the server, even though the probability of this event is fairly low. Hosts that have been inactive for a while may be deleted from memory, thereby saving resources at the relatively small expense of re-activating a host whenever new URLs for the host arrive. During the re-activation process, data structures are re-initialized based on data stored persistently on disk.

5.2.1.1 Traversal component

From among the URLs in the URL queue, the next element scheduled to be downloaded is chosen by the traversal component. Possible ways of traversing the web have been described in Section 2.1. Simple adaptations of these algorithms, however, are inapplicable to and inefficient for web crawling because they restrict parallel access and performance-tuning. For instance, a strict breadth-first traversal would often result in querying a host for a large number of pages in succession (the average ratio of URLs referencing an external host on a web page is 10%), which would generate high demand for and run the risk of overloading a single host. It is more practical to use a hybrid algorithm that allows multi-threaded access and offers load balancing properties while preserving the major characteristics of the original traversal strategy.

The traversal component has one or two inputs and a single output. The URL distributor component (Section 5.2.4) is continuously informing the traversal component regarding the changes in the URL queue, based on which the component maintains the total and the host-specific length of the queue. The load balancer (Section 5.2.1.2) sends

[1] The URL queue (as will be elaborated later in this chapter) is a disk-resident, memory-cached data structure unrelated to the connector FIFO queue used by the component architecture for data transmission. In fact, the details of data exchange will not be emphasized in this chapter as transparent, automated message-based communication has been the very reason for designing the underlying component architecture.

a host-specific ready message to the traversal component when the host can again receive requests after a specific time has elapsed since the last document retrieval. In addition, the system can be configured to allow multiple requests per host, which is often desirable for crawling smaller domains. The use of a load balancer is optional, whenever it is absent, the traversal component has a single input only. (Figure 5.2)

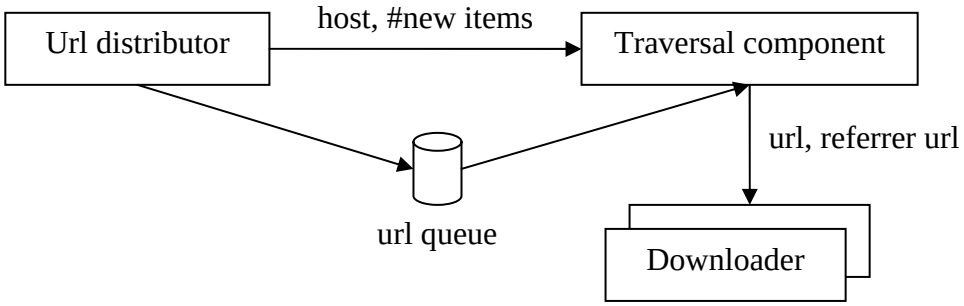

Figure 5.2: Using a traversal component without a load balancer.

Whenever a host is ready to receive requests and the URL queue has items corresponding to the host, the next item is chosen according to the traversal strategy. However, the URL is not immediately forwarded to downloaders. If it were, the traversal component could easily block if the downloaders are busy retrieving documents and can accept no new items, which would prevent the traversal component from accepting input, eventually leading to a deadlock. To remedy the situation, the traversal component is an asynchronous component and hence realizes a pull rather than a push model. Items are fed to the downloaders on-demand, i.e. whenever free capacities arise.

The current implementation includes two selection strategies. The inheritors of the `SimpleTraversal` class such as `ParallelBfs` have a single method that selects the next downloadable page. However, the inheritors of `LoadBalancedTraversal` class need to implement additional methods. Besides selecting the next URL, they must keep track of the changes in the set of the available hosts. The class hierarchy of traversal components is seen in Figure 5.3.

5.2.1.2 Load balancer component

Various approaches exist to prevent overloading any single host with too many requests. One of the most simple approaches is random traversal. If the next document to download is chosen from the URL queue at random, requests to large web sites will be more common while small ones will receive fewer requests. On average, the approach shows the expected behavior but no guarantees against overloading are provided. In particular, when using priorities (see Section 5.2.4.1) the choice of the next URL does not follow a

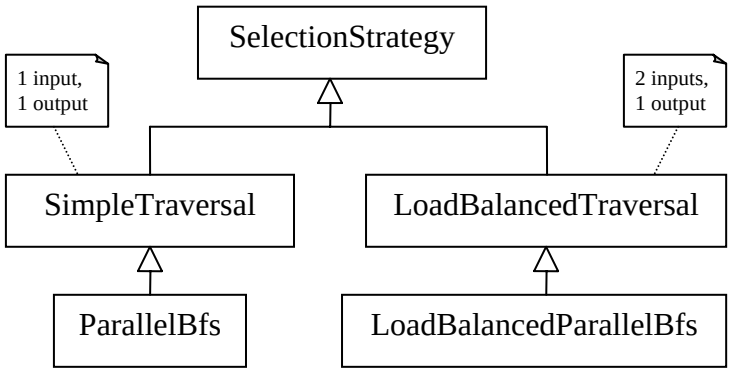

Figure 5.3: Class hierarchy of traversal component types.

uniform distribution, which results in a greater probability of temporary server overload.

In order to provide a strong guarantee that a single request arrives at a remote host at a time and these requests are spread apart with a minimum duration, a separate dedicated component is required. The frequency of requests to a host is externally configurable in the application XML setup descriptor. Optionally, the number of parallel downloader threads can be set instead of the request frequency. This is especially useful when only a few hosts rather than a large proportion of the web is crawled such as when archiving host contents. If simultaneous document retrieval from hosts were not allowed, the crawler would be idle most of the time, waiting a relatively long period for a single access. For a large crawl, this issue does not arise, the number of hosts is large enough to find hosts that are eligible for access.

The scheduling process including load balancing is shown in Figure 5.4. Once a document retrieval operation is finished, the downloader component signals the event to the load balancer. The balancer waits until the host is again ready for a new request and sends a message to the traversal component to select the next URL for retrieval. When an empty slot becomes available in the input queue of the downloader components, the traversal component feeds the URL into the queue. Nonetheless, even if the downloaders are incapable of handling any more requests at the moment, the traversal component still processes messages regarding URL queue length changes and host availability.

Apparently, load balancing is strongly tied to choosing the next URL to retrieve. On one hand, it is pointless to select a URL for retrieval unless its corresponding host is capable of receiving the request. On the other hand, the load balancer has to known when downloading a document from a given host has started and finished. This can only be ensured by a synchronous call from downloaders to the balancer, if this information is passed asynchronously as messages, the order of the messages may incidentally swap, which is a highly undesirable behavior. However, while synchronous calls would be an

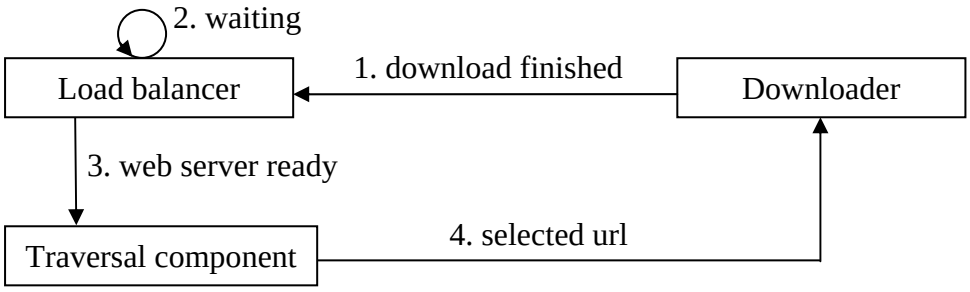

Figure 5.4: Steps of the scheduling process.

ideal solution when the two communicating components are in the same process, when they are distributed across different machines, network latency can become considerable. As a compromise, the load balancer, once it sends a message to the traversal component that a host is ready to receive a new request, it assumes that a new download targeting the host is immediately started. The download statistics the load balancer receives from downloaders contains the exact time the retrieval operation was started and finished.

5.2.2 Downloader component

Downloader components are responsible for retrieving documents corresponding to the URLs they receive. This process includes address resolution, which involves determining the IP address associated with a host name. Address resolution is performed by a DNS resolver provider, which also caches recent results to increase performance.

An important aspect of document retrieval is the compilation of the HTTP header. [20] In particular, this includes setting the `User-Agent` field so that it identifies our crawler (or possibly mocks another popular browser). The `Referer` [sic] HTTP field allows passing the referencing web page when downloading a document; this information is required to traverse a few hosts and is received along the URL by the component. HTTP cookies set by the host may also have to be submitted with each request, the component handles this automatically. Request headers are compiled by a provider with the `IHttpHeaderProvider` interface.

Unfortunately, the .NET framework implementation of the `HttpWebRequest` class [1] is not adequately flexible and is hence unsuitable to our needs. In particular, it does not permit circumventing host name resolution: a separate DNS request is generated for each HTTP request. Even though DNS results are cached locally, the size of this cache is not of sufficient size. Our implementation provides a much faster and scalable solution and can establish connections by itself (re-)using raw IP addresses. The `HttpConnection`, `HttpHeaders`, `HttpRequest` and `HttpResult` classes our system contains not only con-

sume fewer network resources but they also expose a higher degree of versatility. Clearly, the implementation included in the framework offers a general solution and is not intended for the particular field of use.

Once a document has been downloaded, downloader components inform the load balancer component that the operation has finished with statistical information on the size of the document, time elapsed, etc., which makes dynamic per-host performance tuning possible. The downloaded documents are forwarded to parsers.

5.2.3 Parser component

Parsers receive downloaded documents and may serve as front-ends to search engine indexers. While the current implementation is solely responsible for hyperlink extraction, a more sophisticated parser component (or a chain of parser components) can offer subtle tokenization services.

Although it may seem deceivingly simple, hyperlink extraction can involve difficulty due to the diversity of the ways URLs occur in a document. Hyperlinks may be of several kinds: simple unformatted URLs in the text, URLs in `href` and `src` attributes of HTML tags, redirection URLs in `meta` tags in the HTML header, hyperlinks in stylesheets, in JavaScript code, etc. Whenever the parser component comes upon relative URLs, it has to expand them to absolute URLs by prefixing the URL with the host name, the document location or the contents of the HTML `base` tag. [3] Though many types of URLs exists, following hyperlinks on stylesheets and in JavaScript code is often not as rewarding as simple types and consumes significantly more computing capacity.

Before parsing content that is neither textual, nor an HTML-variant, filters are required to convert the document in question to a format the parser component comprehends. IFilter interface filters [4] can handle this job.

Currently, the parser component relies heavily on an external COM component, the MSHTML parser to do its duty. In the future, we plan to replace this external component with a regular expression or SAX [7] parser.

5.2.4 URL distributor component

The URL distributor component is a common front-end to URLs either (1) assigned by the server or (2) extracted from documents downloaded directly by the client. The URL distributor component classifies URLs according to their domain and optionally places them into the URL queue or transmits them to the server for designation to another client.

Web pages scarcely have a single outbound link to another page. In general, they

reference multiple other pages, which means that the number of documents to download increases in an exponential fashion for most traversal strategies. For this purpose, the data structure realizing the URL queue must be stored on disk but it is imperative that the queue be partially cached in memory for faster access. The cache mechanism should depend on the traversal algorithm. For breadth-first traversal, the least-recently inserted elements; for depth-first traversal, the most-recently inserted elements should be cached. Data access to the URL queue is facilitated by a provider. The major advantage of the construction is transparency: by switching the provider both the data source and the cache mechanism may be changed. Moreover, connecting two providers in series corresponds to a layered architecture, with each layer replaceable on need. The lower layer, i.e. the one that does not directly connect to the component, accesses the data source, while the upper layer implements the caching mechanism. (Figure 5.5)

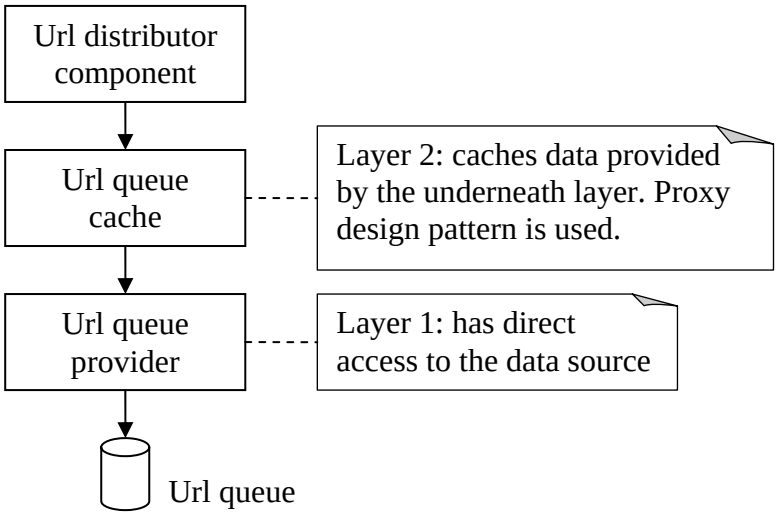

Figure 5.5: Accessing the URL queue by means of layered providers.

In the current implementation, the URL queue is realized by a relational database table. The primary advantages of the approach are simplicity, flexibility and error tolerance. If the system has to restart, only memory-resident data is lost and the structure of the URL reference graph is not corrupted owing to the ACID properties relational databases guarantee. The most crucial drawback is the speed penalty. In further phases of development, we plan a transition to a structured file, which eliminates the overhead of database access and by a full control of the storage mechanism, disk seeks per reads and writes may be significantly reduced.

The URL distributor component performs the following tasks:

- It determines if the client is responsible for processing the URL or it should be forwarded to the server for assignment.

- If the client is responsible for the URL, the component verifies that the URL is not already present in the URL queue. Unless a match is found, the URL is appended to the queue.
- The component determines the priority level belonging to the URL. Classification is performed by a provider.

5.2.4.1 URL priorities

In our system, three levels of priority are distinguished:

- *High priority* is associated with documents that should be downloaded first from their respective host. In other words, other documents should not be downloaded before all high priority URLs have been retrieved. Typical examples are robot exclusion files. Downloading should not proceed until it is discovered which directories on the given host are not subject to retrieval. Another example could be guided or focused download. A user may want to download images, in which case he prefers to retrieve images as soon as possible. By setting higher priority to images, a given number of images are collected more quickly than they would be by resorting to conventional unprioritized breadth-first traversal.
- *Normal priority.* Once all high priority URLs have been retrieved, the system commences with downloading normal priority URLs according to the selected strategy.
- *Skip.* URLs may be exempt from being downloaded in several cases:
  - The URL references a resource that resides in a directory that is restricted by robot exclusion rules.
  - The URL does not satisfy a user-defined domain or URL filter. (See Section 5.3.2)
  - Distance from the first document retrieved from the host exceeds the user-defined limit.
  - The document MIME-*type* is unrecognized, which prevents further processing.
  - The URL protocol is unknown.

5.3 Server components

As previously noted, server components coordinate clients by receiving URLs clients are not expected to process and forwarding these URLs to the respective client already in charge of the host identified by the URL or a newly assigned client if no such exists. URL forwarding

is a dynamic operation in the sense that the recipient of the URL depends on the URL itself. In contrast, the component architecture only provides static interconnection. Once an item is placed into the local part of a remote connector, it is scheduled for remote transmission and in time appears on the remote side. The target of the transmission depends on the configuration and not on the content.

In order to tackle content-dependent item designation, the server features a mass communicator component that manages a large number of remote connections in parallel. The communicator component is seen as the remote constituent of a remote connector from the perspective of clients in the sense that it acts as the server-side chunk of the client-side remote connector. The communicator tags items based on the client connection identifier and feeds them into a single connector queue. Items are extracted from this queue by the marshaler component that assigns a client to each item it receives and places the re-tagged items into another queue. The communicator reads items from this latter queue and forwards them to clients based on the identifier.

5.3.1 The marshaler component

Marshaler components are located in the server and are the primary agents of client coordination. They are single-input-connector, single-output-connector components (instances of `SimpleFilter`) with three providers. Marshalers receive URLs and *designate* URLs to clients. URL designation is a multi-step process.

1. The marshaler component fetches a URL from its input queue.

2. The URL is tested against a *recently seen table*, which is realized as a provider.

   The recently seen table is a fixed-size hash table stored in memory that contains URLs processed by the component not long before. In other words, if a URL is present in the data structure, steps have recently been taken to forward the URL to the respective client. When the URL is present in the table, it is discarded. When it is not found, it is inserted into the table.

   Being fixed-size, the hash table could outgrow its dimensions. To prevent this situation, a reference bit is maintained for each element of the table. When an element is checked for presence, the reference bit is updated (set to 1). In case the hash table reaches a certain predefined saturation factor, elements without a reference bit set are erased. Once these elements have been removed, the reference bit is cleared (set to 0) for each remaining element in the data structure. This means that all these elements are scheduled for deletion unless they are referenced before the saturation factor rises above the prescribed limit again. [16]

The recently seen table can be very effective because URLs to be marshaled follow a Zipfian distribution. In other words, a large number of web pages link to a small subset of the entire web. The recently seen table can filter out commonly referenced URLs without forwarding it to clients for processing, thereby saving considerable bandwidth. A table of 10 000 to 100 000 URLs can significantly reduce communication overhead. [18]

3. The host name part of URL is canonicalized. IP addresses are resolved into host names.

   A host can have multiple names yet a single name is *canonical* and others are referred to as *aliases*. Aliases can lead to downloading the same set of documents multiple times because the URLs appear to be different because of the host name yet they map to the same IP address and same server. URL canonicalization circumvents this phenomenon. Note that this process does not eliminate duplicates due to mirrors.

4. Domain constraints are verified. URLs with host names that do not satisfy domain constraints are discarded.

5. It is verified if the host part of the URL is already assigned to a client. This is a database-backed operation implemented in a provider which returns the globally unique identifier of the client if an assignment exists.

6. Unassigned hosts are assigned to a client based on the assignment algorithm. The assignment algorithm can range from simple random assignment to more sophisticated geographical or domain-based assignment. The algorithm is realized by means of a provider.

7. The URI is extended with the assigned client identifier and the resultant designated URI is placed into the output connector queue.

   Designated URLs are instances of the generic class `DesignatedItem<T>`, which associates an item of type `T` it encapsulates with a unique identifier. In this scenario, the identifier corresponds to a client identifier and the encapsulated item to a URL.

5.3.2 Domain constraints

Domain constraints are an extremely simplified variant of regular expressions [21] that aim to test if a host name (or domain) is a descendant of a(n other) domain. Just as domain names, constraints are hierarchical where parts are separated by dots. For instance, the `org.wikipedia.en`[2] host is a direct descendant (subdomain in DNS terminology) of the

[2]In this section, we use the reverse name both for hosts and for constraints.

`org.wikipedia` domain. On the other hand, `org.wikipedia.en` is an indirect descendant of the `org` top-level domain.

A domain can *match* a constraint in one of the following ways:

- *Exact match.* A case-insensitive comparison of the domain and the constraint is performed.
- *Subdomain match.* When the constraint ends in `+`, all descendants of the constraint match but the domain name that corresponds to the constraint does not. For instance, `hu.bme.+` matches `hu.bme.aut` but not `hu.bme`.
- *Descendant match.* In case the constraint ends in `*`, all descendants of the constraint and also the constraint itself match. For instance, `hu.bme.*` matches both `hu.bme.aut` and `hu.bme`.

Domain constraints can either be *affirmative* or *negative*. Affirmative constraints tell the marshaler component when to forward the URL to a client for processing. For instance, the host name `org.wikipedia.en` does not match the affirmative constraint `hu.*` and is discarded. Negative constraints behave the opposite way: a host name does not meet constraints in effect if it matches any of the constraints. For instance, the aforementioned host, `org.wikipedia.en` satisfies the negative constraint `hu.*` because it is in the `org` rather than the `hu` primary domain. A crawler that collects only pages in the `hu` domain has an affirmative domain constraint `hu.*` in effect without any negative constraints.

5.3.3 The mass communicator component

The mass communicator component acts a server-side stub of multiple client-side remote connectors simultaneously. For each client, it instantiates a `ReceiverProxy` and a `SenderProxy` object, both of which represent a connection to the remote client. When a client-side remote connector transmits a data packet to the server, the `ReceiverProxy` object intercepts the packet, extracts items it contains, tags them with the client connection identifier (i.e. creates `DesignatedItem<T>` instances) and feeds them into the output connector queue of the mass communicator. In contrast, whenever items are available in the input connector queue of the communicator component, the item is forwarded to the appropriate `SenderProxy` based on the item tag. The `SenderProxy` wraps the items in data packets transfers them to the respective client.

The communicator component is resilient against temporary failures. Whenever the connection to a client is lost, items are stored in a temporary buffer (or on disk if data outgrows the size of the buffer) until the connection is re-established. If the failure is permanent, items are fed back into the output queue of the communicator for re-assignment.

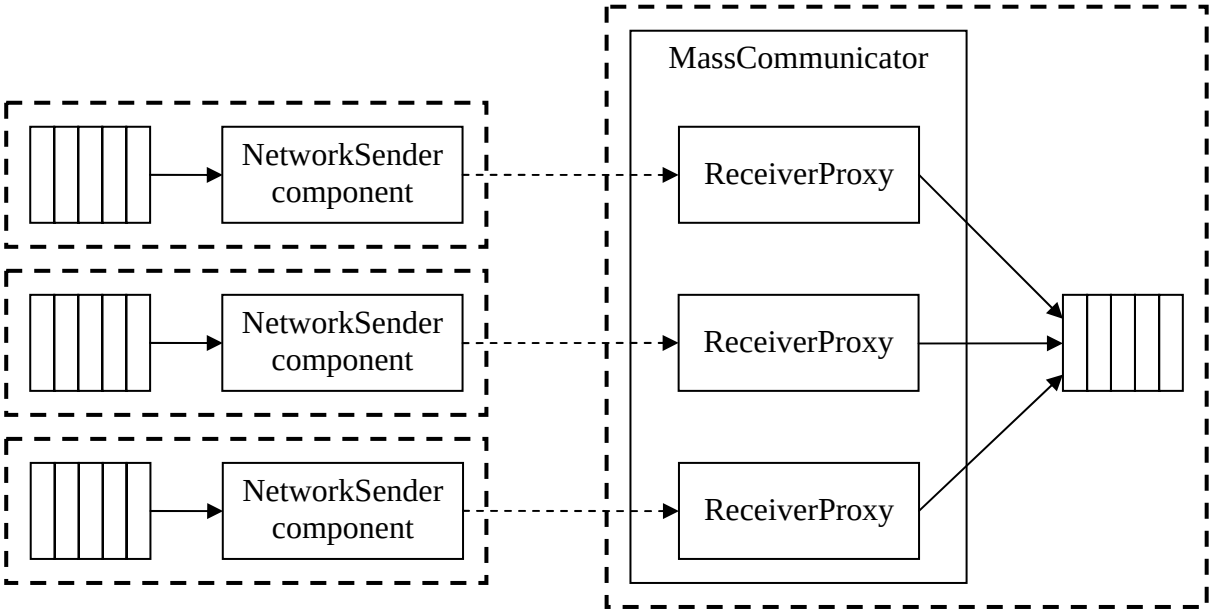

Figure 5.6: The mass communicator component is seen from the client-side as the server-side constituent of a remote connector. Dashed rectangles indicate process boundary and dashed arrows correspond to serialized communication. (Compare to Figure 4.2)

Chapter 6

Evaluation

By having introduced a distributed component architecture in Chapter 4 and by creating a crawler reference implementation based on the architecture in Chapter 5, we have realized an extensible distributed crawling system which possesses both the features scalability and easy management. In this concluding chapter, we compare our crawler to related work, show some dynamic properties of our system, summarize our research and lay out perspectives for future work.

6.1 Comparison to related work

Table 6.1 assesses the system we propose to related work we have introduced in Chapter 3 according to some characteristic criteria:

- We distinguish between *centralized* and *distributed* crawler architectures.
- *Configurability* refers to the degree the crawling system can be fine-tuned by setting properties at startup or by changing configuration files.
- A crawler is *fully extensible* if future functionality can easily be incorporated into the system without disturbing the way other parts of the system work and without any recompilation.
- *Pluggability* identifies the smallest replaceable unit in the system to extend, change or influence crawling behavior. Agent-level pluggability is the coarsest unit and corresponds to the entirety of an independent crawling program or agent, i.e. no real replacement capability. Component-level pluggability allows finer control and refers to replaceablility of subprograms or groups of classes. Class-level replacement is the finest level of control catering for the pluggability of classes behind well-defined interfaces. A diverse model combines all levels of pluggability.

Table 6.1: Comparison of the major characteristics of four crawler systems.

Criterion	**Mercator**	**PolyBot**	**UbiCrawler**	**Crawler.NET**
Architecture	centralized	distributed		
Configurability	high	medium	low	high
Extensibility	supported	limited	not supported	supported
Pluggability	classes	components	agents	diverse
Target environment	single machine	LAN	not restricted	
Scalability	limited		not limited	
Communication volume	n.a.	large	low	medium
Default traversal	breadth-first variant		special[a]	breadth-first
Host access control	single thread	data structure	single thread	data structure
Data storage	centralized		distributed	
Fault tolerance	medium	medium	highest	high
Language	Java	C++, Python	Java	.NET

[a]Implements a host-specific breadth-first traversal algorithm.

- *Target environment* refers to the environment in which the crawler has maximum potential performance.
- *Scalability* is *not limited* unless internal bottlenecks exist in the system that cannot be offset by replication. It is *limited* if it does not scale beyond a point due to disk speed requirement, insufficient computing resources, etc.
- In *single-threaded host access,* data retrieval is performed from a thread exclusively dedicated to the host. If threads are not assigned to hosts, shared *data structures* are used to prevent (or limit) concurrent access.
- Crawlers with *centralized data storage* build a single URL reference graph. Those with *distributed data storage* maintain independent, possibly implicitly synchronized databases.
- If *fault tolerance* is *medium,* the system survives restarts. For *highly* fault-tolerant architectures, the system remains operational even if some of its building blocks are temporarily down. *Highest* fault-tolerance corresponds to uninterrupted operation in the case of permanent errors.

Table 6.2: Comparative connector transmission times.

Connector type	Local	Remote							
Size of transmission batch	1	1	5	10	50	75	100	500	1000
Transmission time in μs	12	921	628	419	408	359	359	372	367

Table 6.3: Comparative performance of various unit types.

Action	Time taken (μs)	Throughput (1/s)
Calling a provider method	6	160000
Transmitting an item through a connector	12	51406
Processing an item by a `SimpleFilter`	32	31296
Processing an item by a `ComplexFilter`	58	17227

6.2 Dynamic properties

Although the scalability of our architecture relaxes the strain on computing resources, communication delay and the relative overhead of our framework might still be of interest. In a series of performance benchmark tests executed on an AMD Athlon64 3000+ running Microsoft Windows XP Professional SP2, some comparative metrics have been obtained.

Table 6.2 shows the latency time of connector message transmission for both local and remote cases. In this test, 10 000 random URLs were generated, fed to and extracted from a connector and time required for the transmission measured. In order to exclude network latency from the total delay, remote connectors were set up in a loopback scenario, the local and remote parts of a remote connector both run in the same process. Additionally, in the case of remote connectors, the size of the batch in which URLs were sent was also varied.

Table 6.3 shows the processing overhead and throughput of different architecture building block types. Calling a provider method is an inexpensive operation because it involves a pure method call and pass-by-reference. Transmitting an item through a local connector is a more costly operation due to the required thread synchronization and data structure maintenance. Indeed, passing an item through a connector involves placing the item into and retrieving the item from a temporary buffer. When evaluating the performance of components, input and output connectors were bound to the component and the characteristics of the resultant subsystem was measured. A `ComplexFilter` incurs a larger overhead than a `SimpleFilter` because of the reflection services it employs, which cater for the greater flexibility of this component type (e.g. multiple inputs, automatic setting and retrieval of property values, etc.).

Nonetheless, none of the performance metrics of the component architecture are actual limitations. The creators of [28] used a configuration with a speed of about 140 pages per second, which accounts to a speed of 12 million pages per day, which they believe suffices for most academic projects. Additionally, they calculate with a speed of 70 to 100 pages per second for each node (machine) of a high-speed crawler they construct by increasing the degree of distribution in their system. The constraints of our architecture are far above these values.

6.3 Summary

Despite the deceptively simple crawler algorithm, we have seen that the algorithm in its naïve form does not scale to the enormous size of the Internet. In spite of the use of multiple concurrent threads, disk-seek minimizing data structures and efficient caching mechanisms, a centralized architecture is insufficient to battle the resource demand of the almost-exponentially growing web. A distributed, scalable architecture can, however, face the required computation complexity a large-scale crawl poses by splitting not closely integrated tasks across separate machines and replicating distributed functionality. Nevertheless, a distributed architecture is significantly harder to coordinate and escalates required developer effort.

The architecture we have proposed targets flexibility in managing a distributed architecture, thereby harnessing the potential in scalability while catering for easy development. Components, which are the primary building blocks of the application, encapsulate integrated replicable functionality. Data flow between components is an asynchronous, message-based communication, in which messages themselves are temporarily stored or transparently transmitted over the network by connectors. Providers are a means of accessing non-replicable functionality such as databases or sited (machine-specific) resources. The implementation of components and providers is orthogonal to the way components are interconnected; the framework provides a declarative way of description of how individual components are spread across machines. Thus, the system is loosely coupled, easily configurable and extensible.

In order to prove that the architecture we have introduced is sound, we have implemented our own distributed web crawler. It is a coordinated crawler consisting of clients and one server (or possibly more servers). The distinction between server and client is only conceptual, both the server and clients themselves consist of multiple components and are therefore distributable across several machines. Clients are, at minimum, comprised of URL distributor, traversal strategy, load balancer, downloader and parser components and are freely extensible with document filters, index builders, statistics compilers or im-

age processors. Each client is associated with an own URL queue that lists all URLs the client is to visit and a URL reference graph that stores how hyperlinks are directed from page to page. The server wraps a mass communicator and a marshaler component, which maintains connections and designates domains to clients, respectively.

Our crawler realizes a dynamic partitioning of the web by the server assigning domains to clients. Clients crawl the domain they have been assigned and transmit foreign URLs back to the server for re-assignment. The system achieves zero overlap, maximum possible coverage and limited client-server communication due to the small proportion of inter-domain compared to intra-domain URLs and global Zipfian distribution of document URLs. While the quality of downloaded pages may be worse than that of a centralized crawler because of transmission delays, this is usually not significant in a large-scale crawl.

While the distinction of server and client is not strict due to the underlying component architecture, placing clients near the domains they are expected to crawl can considerably reduce network load because of domain locality. In fact, our system is flexible enough to be used both on a local network of interconnected machines, on a set of globally scattered machines or a combination of the two.

6.4 Perspectives for future work

While surpasses existing architectures such as [23, 28, 13] in terms of combined flexibility, ease of configurability and required development effort (see Table 6.1), the proposed architecture is not yet comprehensive. Below we list some points of possible future work.

Data management is currently performed by a relational database which does not exploit some special characteristics of the data our system has to handle. For instance, the hierarchical nature of URLs and hosts in particular is not taken directly into account during data storage and could be incorporated only as expensive joins. In fact, strong ACID properties or transaction support databases provide are completely unnecessary in our system as data loss can be easily recovered by a re-crawl. A simple, file system-based structured data storage could suffice which would not incur the database overhead cost. A simple storage would also allow closer integration of caching mechanisms and data storage, which would contribute to a further increase in speed. Note however that replication of clients can offset the speed penalty due to not optimized data access so that this is not a severe limitation in our system.

While our system is resilient to temporary failures as remote connectors can store items if their network connection is lost and transmit data once the connection has been re-established, the single server we have introduced into a system can undoubtedly increase the vulnerability of our system. We plan to improve the architecture by multiple

coordinators that work in parallel to designate URLs to clients.

Although our system has support for dynamic configuration and tracking, this is not currently facilitated by a visual front-end. Even though the current implementation contains a very simple graphical user interface to query component, connector and provider status, this is not yet mature: dynamic reconfiguration through the interface, coordination of server and client components or network flow monitoring are not yet possible, which would be fundamental to in-depth administration.

Additionally, we plan to perform a large-scale crawl first by a simulation of the web based on its statistical characteristics available in research, second by a real crawl. These might reveal some deficiencies currently unseen in the system and will promote future development.

Bibliography

[1] *.NET Framework Class Library Reference*, Microsoft Visual Studio 2005 Documentation, 2005

[2] *Apache HTTP Server Version 2.0 Documentation*, The Apache Software Foundation, 2005

[3] *HTML 4.01 Specification*, W3C Recommendation, 24 December 1999, `http://www.w3.org/TR/html4/`

[4] *IFilter*, `http://www.ifilter.org/`

[5] *Microsoft Internet Information Services*, `http://www.microsoft.com/iis`

[6] *Microsoft* SQL *Server 2005*, `http://www.microsoft.com/sql/`

[7] *Simple API for XML*, `www.xml.org/xml/resources_focus_sax.shtml`

[8] *SourceForge.net*, `http://sourceforge.net`

[9] *Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/wiki/Main_Page`

[10] *XHTML 1.0 The Extensible HyperText Markup Language* (Second Edition), A Reformulation of HTML 4 in XML 1.0, W3C Recommendation, 26 January 2000 (revised 1 August 2002), `http://www.w3.org/TR/xhtml1/`

[11] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM, Volume 13, Issue 7, July 1970, pp422-426

[12] Bodon Ferenc, *Adatbányászati algoritmusok*, 2006. március 7., `http://www.cs.bme.hu/~bodon/magyar/adatbanyaszat/tanulmany/`

[13] Paolo Boldi, Bruno Codenotti, Massimo Santini, Sebastiano Vigna, *UbiCrawler: A Scalable Fully Distributed Web Crawler*, `http://law.dsi.unimi.it/index.php`

[14] Tim Bray et al., *Extensible Markup Language (xml) 1.0 (Second Edition)*, October 6, 2000, `http://www.w3.org/TR/2000/REC-xml-20001006.html`

[15] Sergey Brin, Lawrence Page, *The anatomy of a large-scale hypertextual Web search engine*, Computer Science Department, Stanford University, Stanford, CA, 1998

[16] Andrei Z. Broder, Marc Najork, Janet L. Wiener, *Efficient* url *Caching for World Wide Web Crawling*, 2003

[17] David Carmona, *Programming the Thread Pool in the* .net *Framework*, Microsoft Developers' Network, June 2002,
`http://msdn.microsoft.com/library/default.asp?`
`url=/library/en-us/dndotnet/html/progthrepool.asp`

[18] Junghoo Cho, Hector Garcia-Molina, *Parallel Crawlers*, WWW2002, Honolulu, Hawaii, May 7-11, 2002, ACM 1-58113-449-5/02/0005

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, Second Edition, The MIT Press, 2001

[20] R. Fielding et al., *Hypertext Transfer Protocol – HTTP/1.1*, Network Working Group, RFC 2068, January 1997

[21] Jeffrey E. F. Friedl, *Mastering Regular Expressions*, 2nd Edition, O'Reilly

[22] John Erik Halse et al., *Heritrix developer documentation*, Internet Archive, `http://crawler.archive.org/articles/developer_manual/index.html`

[23] Allan Heydon, Marc Najork, *Mercator: A Scalable, Extensible Web Crawler*, Compaq Systems Research Center, Palo Alto, CA

[24] Martijn Koster, *A Standard for Robot Exclusion*, 30 June 1994, `http://www.robotstxt.org/wc/exclusion.html`

[25] P. Mockapetris, *Domain names – Concepts and facilities*, Network Working Group, RFC 1034, November 1987, `http://tools.ietf.org/html/rfc1034`

[26] Dave Raggett, Arnaud Le Hors, Ian Jacobs, html *4.0 Specification*, April 24, 1998, `http://www.w3.org/TR/1998/REC-html40-19980424`

[27] S. Shepler et al., *Network File System (NFS) version 4 Protocol*, Network Working Group, RFC 3530, April 2003, `http://tools.ietf.org/html/rfc3530`

[28] Vladislav SHKAPENYUK, Torsten SUEL, *Design and Implementation of a High-Performance Distributed Web Crawler*, Proceedings of the 18th International Conference on Data Engineering, CIS Department Polytechnic University, Brooklyn, NY, 2002

[29] SZEREDI Péter, LUKÁCSY Gergely, BENKŐ Tamás, *Theory and Practice of the Semantic Web* (to be published in 2007, translation based on: SZEREDI Péter et al., *A szemantikus világháló elmélete és gyakorlata*, TYPOTEX, Budapest, 2005)

[30] Eric W. WEISSTEIN, *Red-Black Tree*, MathWorld – A Wolfram Web Resource, `http://mathworld.wolfram.com/Red-BlackTree.html`

[31] I. H. WITTEN, A. MOFFAT, T. C. BELL, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Second Edition, Morgan Kaufmann, 1999