Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Crawler.NET:

Coordinating and controlling distributed web robots

Elosztott webrobotok koordinációja és vezérlése

Készítette:
Hunyadi Levente
hunyadi.levente@enternet.hu

Konzulensek:
Gincsai Gábor, doktorandusz-hallgató
Dr. Levendovszky Tihamér, egyetemi adjunktus
Automatizálási és Alkalmazott Informatikai Tanszék

2007

Nyilatkozat

Alulírott Hunyadi Levente, a Budapesti Műszaki és Gazdaságtudományi Egyetem szigorló hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem. Tudomásul veszem, hogy az elkészült diplomatervben található eredményeket a Budapesti Műszaki és Gazdaságtudományi Egyetem, a feladatot kiíró egyetemi intézmény saját céljaira felhasználhatja.

Budapest, 2007. május 18.

Hunyadi Levente
szigorló hallgató

Összefoglaló

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.

Ahhoz, hogy a dokumentumhalmaz bejárása hatékonyan történjen, azaz a gyakran frissülő oldalakat kellő gyakorisággal látogassuk meg, elengedhetetlen a párhuzamosítás. Ugyanakkor egy számítógépeken átívelő, párhuzamos végrehajtást támogató elosztott rendszer működtetése lényegesen összetettebb, mint egy központosított rendszeré. További kulcsfontosságú aspektusok a könnyű nyomonkövethetőség és a rugalmas szerkezet. Elengedhetetlen, hogy a kialakítandó rendszer támogassa a feladatok végrehajtásának figyelését, valamint könnyen alkalmazkodjon eltérő feladatokhoz is (pl. konkrét tartományok bejárása, nyelvspecifikus viselkedés, hivatkozási szerkezet-vizsgálat).

A felsorolt elvárásoknak eleget tevő rendszer kialakítására a szerző egy olyan alaparchitektúrát mutat be, amelyet egymáshoz lazán kapcsolódó, jól definiált felületekkel illeszkedő, üzenetekkel kommunikáló egységek alkotnak. A laza csatolás lehetővé teszi, hogy az egyes egységeket különböző számítógépeken futtassuk, másrészt támogatja új funkcionalitás beépíthetőségét. Az egységek közti helyi vagy távoli kommunikációt a rendszer átlátszó módon kezeli, az egységek összekapcsolásának módja pedig XML állományok segítségével deklaratív módon írható le, mindez elrejti az elosztott rendszer koordinációjával kapcsolatos teendőket és csak saját feladatukra koncentráló önálló egységek fejlesztését teszi lehetővé.

Az architektúra működését a szerző egy referencia-implementációval demonstrálja, amelyben egy vezénylő komponenshez számos ügynök kapcsolódik. Mind a vezénylő, mind az ügynökök számos, akár több számítógépen futó egységből áll-

nak. A vezénylő feladata a web particionálása: egy-egy szeletét egy-egy ügynöknek utalja ki. Az ügynökök feltérképezik a feladatul kapott kiszolgálót vagy tartományt, a külső hivatkozásokat pedig visszaküldik a vezénylőhöz. Mivel az ügynökök tipikusan a bejárandó tartományokhoz földrajzilag közeli helyen működnek, a külső hivatkozások részaránya pedig jelentősen kisebb, mint a belső hivatkozásoké, jelentős hálózati forgalom-csökkenés érhető el. A rendszer működésének figyelését egy grafikus alkalmazás támogatja, amely lehetőséget biztosít az aktuális paraméterek lekérdezésére, esetleges módosítására.

Az alkalmazás megvalósítása a .NET keretrendszerre alapul, erőteljesen támaszkodik annak szálkezelési és távoli kommunikációbeli építőelemeire. A rendszer forráskódja C# nyelven készült, az adatműveleteket a flexibilitás érdekében adatbáziskiszolgáló végzi, amely területen a Microsoft SQL Server megvalósítására esett a választás, ez azonban a hatékonyság további fokozása érdekében hasonló illesztőfelület mellett egyszerű struktúrált állományra cserélhető, az implementáció 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.

For the traversal to be effective, that is, frequently updated documents to be visited regularly, parallelization is indispensable. However, operating a distributed system that supports parallelization across multiple machines is a substantially more complex job than operating a centralized system. Other crucial aspects include easy tracing of job completion and flexible architecture. It is essential that the system support monitoring how jobs are processed and it adopt to various crawling tasks (e.g. traversal of specific domains, language-dependent behavior, reference structure analysis).

In order to battle the aforementioned demands, the author presents an architecture comprising of units loosely-coupled by means of well-defined interfaces and communicating with one another by exchanging messages. Loose coupling allows units to run on different machines and caters for extension with new functionality. Communication between units is transparent independently of whether it is in fact a local or a remote message exchange, and the way units are interconnected is declaratively defined in XML descriptors. In general, the services offered by an underlying framework greatly reduce the complexity that would otherwise be related to coordinating a distributed system. Unit development may subsequently focus on other aspects of web robot construction.

The way the architecture operates is demonstrated by a reference implementation in which multiple agents connect to a coordinator component. Both the coordinator and the individual agents may themselves consist of units distributed across multiple

computers. The coordinator partitions the web and maps a partition to a crawling agent. Agents traverse the hosts and domains they have been assigned sending external references back to the coordinator. As agents are typically geographically close to domains they are to explore and the ratio of external to internal links is adequately low, substantial savings on network traffic may be attained. A graphical user interface helps monitor the system, allowing querying status and adjusting behavioral parameters.

The application is implemented in the .NET platform and fully utilizes thread management and remote communication facilities the platform provides. The source code of the framework has been implemented in C#, while data operations are performed by a relational database for the sake of flexibility. Though the particular implementation uses Microsoft's SQL Server, this can be replaced with a structured file for further efficiency gain as the implementation moderately exploits the power SQL offers.

Contents

Chapter 1

Introduction

The greatest challenge to any thinker is
stating the problem in a way that will allow a solution.
Bertrand Russell, English logician and philosopher

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. As a result, locating scattered information has become a cumbersome endeavor. Hence, the role of automated *search engines* 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, videos, PDF files, word processor documents, etc. In order to make documents eligible to be returned as search results, their content should be analyzed by search engines and relevant information extracted to compile an *index*. The index maps *terms* (i.e. words and phrases sought) to documents. User *queries* are then evaluated against the index [36].

In order to compile a massive document index, search engines rely on *web crawlers* (also known as *web robots* or *spiders*). Web crawlers traverse the Internet by hopping from document to document following *hyperlinks* they contain. In fact, the operation of a web crawler can be described by the deceptively simple algorithm in Figure 1.1 [34].

Nevertheless, for the index to remain up-to-date, web traversal should be performed efficiently. Unfortunately, single-machine architectures quickly stumble into

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. (Query string parameters are name-value pairs in the part of the URL that follows the `?` mark, 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.

system limits: memory use, hard disk speed and network bandwidth are all serious constraints. Parallelization is a straightforward remedy to overcome these limitations, in which case the job is split up among identically programmed independent participants. A distributed system executing simultaneously on multiple machines, however, involves a higher complexity in terms of communication, co-operation and synchronization.

Despite their complexity in management, distributed crawlers have major advantages in terms of scalability, dispersion of network load and an overall decrease in network traffic compared to centralized architectures. Computing and data storage capacity can adapt to the demands of the job by adjusting the number of workers (scalability), crawling agents can be placed at locations traffic-wise near the web domains they are to process (network load dispersion), thereby also requiring less data to be transmitted over the network during a crawl (network traffic reduction) [19].

Even though web crawlers are widely employed, 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 [25] or do not lend themselves to easy configuration. [33, 14] A web crawler that harnesses the scalability potential in distributed systems whilst providing ample support for pluggable components and configurable behavior can act as a front-end for further research of search engines, web content or structure mining.

1.1 Scope and structure of this paper

In this thesis, the author presents a novel architecture for implementing a distributed web crawler. The proposed system comprises of independent *components,* which are loosely coupled by means of local or remote *connectors* (the latter conceptually synonymous to FIFO queues). Each component encapsulates a specific task, such as scheduling download requests to servers, downloading documents, parsing documents for hyperlinks or partitioning the web. Connectors transmit data between components and thus allow transparent distribution of components across distributed computers. *Providers,* the third major building block of the architecture, expose machine-local shared resources to components or other providers, such as resolved IP addresses of domain names, recently referenced URLs or database (filesystem) resources.

Both components and providers are fully object-oriented with appropriate base classes providing fundamental services common to all inheritors, especially with respect to initialization, coordination and synchronization. Thus, the intricacy of a distributed architecture is hidden by the base classes, and inheritors may concentrate on crawler-specific tasks. In addition, this design facilitates extension. Components communicate without being aware of the specific characteristics of their communication partners, and providers are pluggable if they expose the same interface. Thus, new functionality can easily be incorporated into the system e.g. by adding a new document parser component or a different traversal strategy provider.

In order to demonstrate the power of the loosely-coupled component architecture, the author outlines a distributed web crawler with a central coordinator. The coordinator (or *master*) is responsible for assigning tasks to agents (*workers*), the latter of which perform the tasks of collecting and analyzing web pages, and build-

ing the URL reference graph. Both the master and workers are distributable across multiple computers and themselves consist of highly configurable components.

The proposed system is implemented in the .NET framework. The choice was driven by the versatility of this platform without a severe speed penalty. In addition, web crawlers realized as part of previous research are implemented in C, C++, Java, Perl or Python and no known distributed crawler is available for the .NET platform. The system described in this thesis is an open-source project under constant development and available for download through anonymous CVS login from the SourceForge.net site [8]. [26] gives a description of a preliminary version of the proposed system.

This thesis is structured as follows. Chapter 2 gives an introduction into concepts relevant to understanding the merits of the component framework and the master-worker crawler architecture. Related work, including the Mercator, PolyBot and UbiCrawler crawling systems, is discussed in Chapter 3. The basic component architecture underlying the crawler is presented in Chapter 4. Chapter 5 outlines the master-worker system and elaborates on the exact components the master consists of and how it interoperates with workers. 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 Chapters 4 and 5 are entirely own contribution. This in particular includes but is not limited to automatized thread management for components, transparent local and remote asynchronous message-based communication, the freely configurable pluggable component model and the graphical user interface for monitoring. 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 general-purpose collection types provided by the .NET framework, which perform poorly in the particular large-data, high-load scenario.

This thesis builds on an intermediate knowledge of application-level web protocols, most emphatically HTTP [21], and SGML languages like HTML [31] or, in particular XML [15]. Throughout the paper but especially in Chapter 4 and 5, the author makes 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 issues are introduced that are essential to implementing a distributed web crawler though did not directly appear in the crawling algorithm in Chapter 1. This includes traversal strategies, the worker thread model, traversal constraints and URL locality. *Traversal strategies* define the principle according to which the next URL to download is chosen. They include the widely known breadth-first and depth-first strategies. We will see how traversal strategy affects the resources a crawler requires and hence how it influences its architecture. Most tasks in a web crawler can be executed in parallel. The *worker-thread model* allows applications to exploit multi-threading without the danger of launching too many threads, leading to the so-called *thrashing;* this model will be explored next. *Traversal constraints* help discard hazardous links following which the crawler can be easily lured into the so-called *crawler traps.* These constraints also facilitate focused crawling and give hint to avoid visiting pages with no valuable or malicious content. The concluding section of this chapter discusses URL and host *locality*, which is fundamental to the operation of any distributed 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 and phrases (called *terms*), which are then inserted into their database as term-to-document pairs [36]. Owing to the rapid rate of change of content available on the Internet, it is practically impossible to have a complete snapshot of the entire Web. Subsequently, an index that maps terms to documents may never be fully

up-to-date. Hence, it is essential that the crawler download important pages first and re-visit quickly changing pages more frequently [34].

In the following sections, some traversal strategies used in web crawlers will be discussed. For the sake of simlicity, the algorithms will be demonstrated with the help of an example graph, in which documents correspond to nodes and hyperlinks to edges (Figure 2.1).

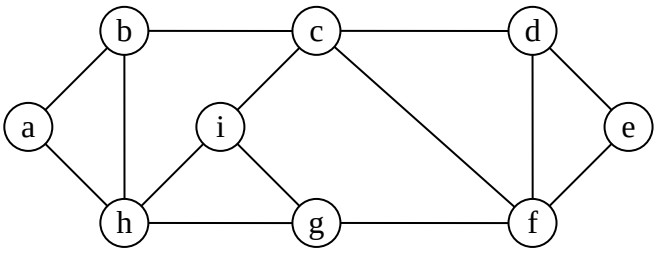

Figure 2.1: Example graph in which the different traversal algorithms are illustrated.

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 seen as nodes and hyperlinks as edges of a graph, breadth-first traversal can be thought of as the advancement of a wavefront in the graph (Figure 2.2). The algorithm starts with a single node behind the wavefront, representing the *seed* document.[1] In the first step, the wavefront engulfs all neighbors of the initial node. Similarly, further nodes directly accessible through edges from nodes just behind the wavefront are engulfed in each iteration. We define the *parent* of a node as the node from which it was first accessed as the wavefront advanced. 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) [20].

2.1.1.1 Application

The breadth-first strategy is commonly implemented by means of a FIFO (first in, first out) queue. As they are visited, neighboring nodes are appended to the end

[1]Note that having multiple seed documents does not influence how the algorithm works. This apparently more complex case reduces to the singe-node case by introducing an auxiliary seed node and connecting all actual seed nodes to the auxiliary node.

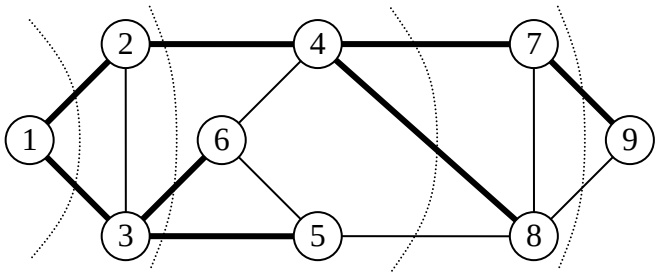

Figure 2.2: Wavefronts and expansion order of nodes in the breadth-first traversal algorithm.

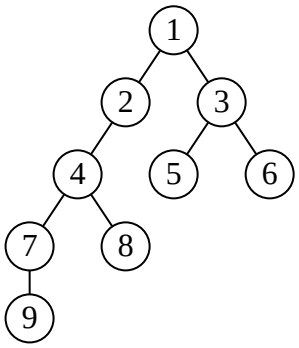

Figure 2.3: Breadth-first spanning tree. Compare this figure to Figure 2.2

of a data structure that can conceptually be seen as a list. Nodes to be visited are extracted from the head of this list (Figure 2.4).

Besides simplicity, the primary advantage of this implementation is easy generalization to multi-threaded environments. With minimal synchronization, threads can simultaneously extract nodes from the head of the list and append unvisited neighboring nodes to the end. In order 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 is not inherited to a multi-threaded environment. Hard 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 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. A thread with more resources and a greater external network bandwidth might access a node quicker

```
def bfs(queue):
    while not queue.empty():
        node = queue.dequeue()
        if not node.mark_processed():
            node.process()
            foreach neighbor in node.neighbors():
                queue.enqueue(neighbor)
```

Figure 2.4: Pseudocode of the breadth-first traversal algorithm implemented by means of a FIFO queue. *mark_processed()* is an atomic operation: in addition to returning with a boolean value indicating whether the node has been marked as visited, it indivisibly sets the marking for unmarked nodes.

than another even if the latter is actually closer to the node w.r.t. the number of clicks from the seed document.

Furthermore, special attention is required when generalizing the algorithm to threads running on multiple computers (multi-agent scenario). The algorithm implicitly assumes a shared knowledge of visited hyperlinks, which is not obviously attained in the case of a distributed architecture. Either the different agents should use a common database that stores information as to which hyperlinks have been visited, or the document set should be unambiguously partitioned so that each agent is responsible for maintaining this information only for a particular domain.

2.1.1.2 Variants

A remarkable variant of the breadth-first strategy is *host-specific breadth-first* traversal. In this scenario, only those hyperlinks are followed by the crawler which correspond to documents on the same host (or in the same domain) as the document which contains them. The primary benefit of this approach is that the hyperlink seen data structure is confined to a host (or domain), 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 [29] shows that a breadth-first strategy tends to download important pages first. This observation is supported by 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 (such as [6] or [12]) 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. 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 [34].

2.1.2 Depth-first traversal

In *depth-first traversal* (also known as *depth-first search* or DFS), the selection strategy for the next node is different than what is applied in breadth-first traversal. While breadth-first traversal advances sequentially with the immediate neighborhood of the current node, depth-first traversal digs deep into the graph. More precisely, the next node the algorithm chooses is always a single 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.

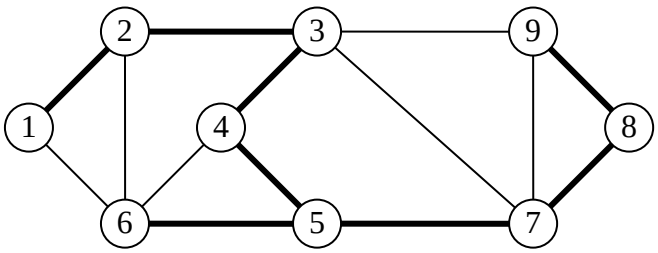

Figure 2.5: Expansion order of nodes in the depth-first traversal algorithm.

2.1.2.1 Application

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 LIFO (last in, first out) queue. [20]

```
def dfs(node):
    if not node.mark_processed():
        node.process()
        foreach neighbor in node.neighbors():
            dfs(node)
```

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

```
def dfs(stack):
    while not stack.empty():
        node = stack.pop()
        if not node.mark_processed():
            node.process()
            foreach neighbor in node.neighbors():
                stack.push(node)
```

Figure 2.7: Pseudocode of the iterative 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 2.3).

In addition, depth-first traversal can be used to fetch entire subsections of the Web easily, such as contents of a host or a smaller domain. DFS is more efficient than BFS in terms of caching because the least recently added node is more likely to be used sooner. In contrast, recently added nodes in a breadth-first traversal are likely to be re-used at only a much later stage as the number of neighboring nodes increases exponentially with each step away from the seed.

2.1.2.2 Variants

Random walk on the document graph can be seen as a variant of depth-first traversal. Here, the next node to be visited is chosen randomly from among the outgoing edges of the current node. One of the key features of this traversal method is that there is no need to keep track of the already visited nodes. Random walk can be used in exploring the structure of the Web, such as discovering clusters or a set of popular (frequently referenced) pages.

2.2 The worker-thread model

In multi-threaded architectures, using a "proper" number of threads is of paramount importance. It is clear that having few threads does not exploit available system resources, while too many leads to the so-called *thrashing*. If the number of threads is below optimal, idle periods are frequent, in which a fast unit, most commonly the processor, is without a job, while each active thread is waiting for a slower unit, such as a hard disk or a network transfer, to complete. On the contrary, a disproportionately high number of threads involves too many context-switching operations, wasting processor resources.

As a logical compromise, the worker-thread model harnesses the power of a multi-threaded architecture with a limited number of so-called *worker threads*. Jobs arrive into a FIFO queue from where they are taken by any of the idle workers (Figure 2.8). Unless a thread in the worker *pool* is idle, jobs are accumulated in the queue. The model is protected against excessive context-switching because the number of active worker threads in the pool has an upper limit, regardless of the number of items in the queue. Should the queue become full, new requests are dropped. The performance of the worker-thread model can be tuned by adjusting the maximum number of threads and an idle period after which worker threads are terminated [18].

Even in its simplest form, the worker-thread model offers a strong guarantee that once a job is placed in the queue, it will be processed in a predictable time. If each request is associated with a newly created thread, in a high-load scenario, threads born by late-comer requests can significantly slow down early-comers. In contrast, the worker thread model always prefers early-comers to late-comers through the job queue.

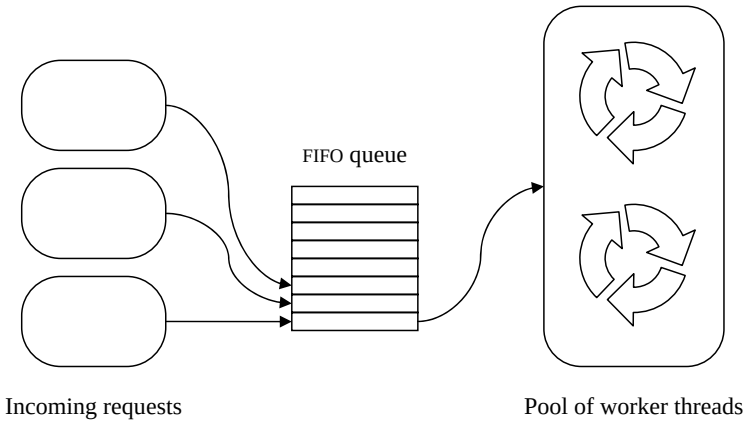

Figure 2.8: The worker-thread model.

2.3 Traversal constraints

During web traversal, a crawler must combat various hazards, known as *crawler traps*. A common variant of crawler traps is 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 [34].

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 [1] 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.

Apparently, 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. For instance, pages of the English Wikipedia project [10] 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* [27] 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 [14] 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 [36] and discard the page without following hyperlinks if it is not written in the target language.

2.4 URL locality

Cho and Molina conducted experiments as to how URL distribution between independent crawler processes affects the volume of inter-process communication [19]. 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.

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:

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

In Equation 2.1, 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 im-

portance metric at a given instant. For instance, the PageRank algorithm [16] defines relevance with the recursive formula:

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

In Equation 2.2, $PR(A)$ 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

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 [19].

Chapter 3

Related work

Strive for perfection in everything you do. Take the best that exists and make it better. When it does not exist, design it.
Sir Henry Royce, British engineer

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 [25] 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,

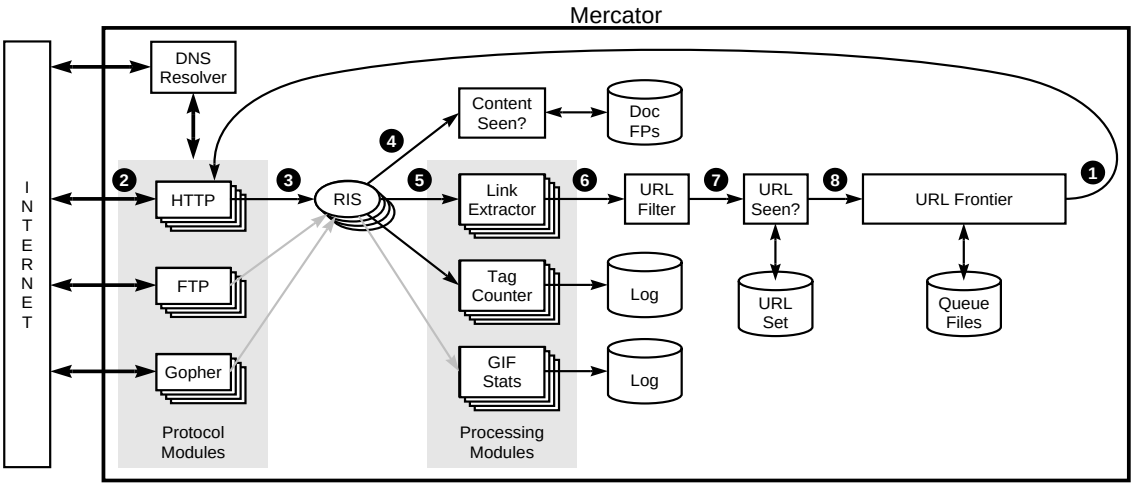

Figure 3.1: The architecture of the Mercator crawler.

- 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 comprises of the following standard building blocks (Figure 3.1):

1. 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.

2. *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 [27].

3. A *rewind input stream* (RIS) 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.

4. After downloading a document, a *content-seen test* is performed to see if the document has been processed before. The content-seen test computes the document fingerprint according to Rabin's algorithm [30] 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.

5. 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 documents for outbound URLs.

6. URL *filters* provide a mechanism to sort out URLs that are not intended to be downloaded before they are forwarded 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.

7. 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 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 efficiently 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.

3.2 The PolyBot crawler

PolyBot [33] 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 for multiple 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

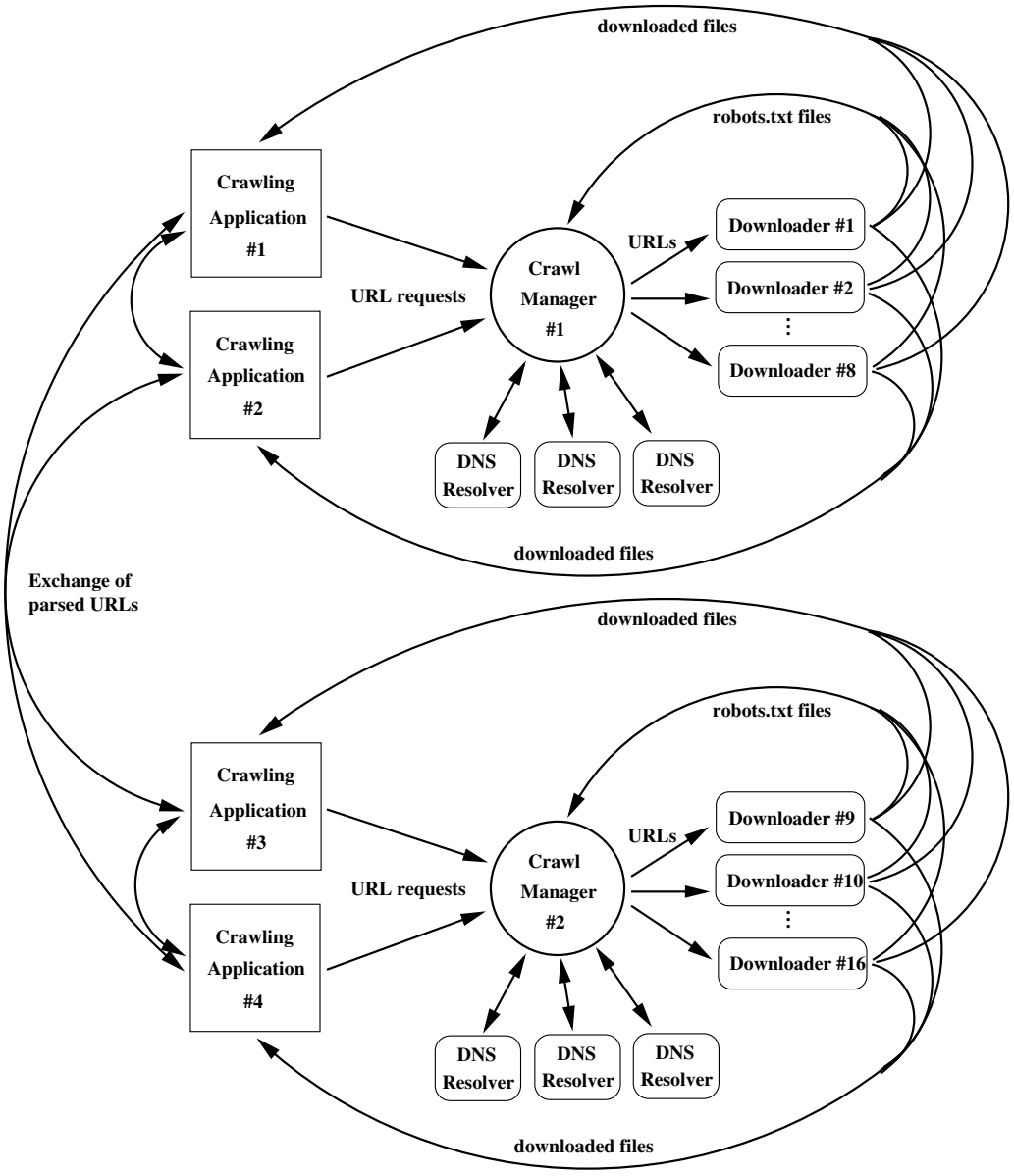

Figure 3.2: The architecture of the PolyBot crawler.

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 exchanging large amounts of data. NFS [32] 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) [22]. Extracted URLs are matched against a URL-seen structure. This structure is implemented as a red-black tree [35] 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.

Just 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 coarse-grained 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.

While NFS offers good performance when run in a local area network, it 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 [14] 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, which 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 of the n buckets 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 bucket to whose replica the hash value is nearest in a clockwise direction (Figure 3.3).

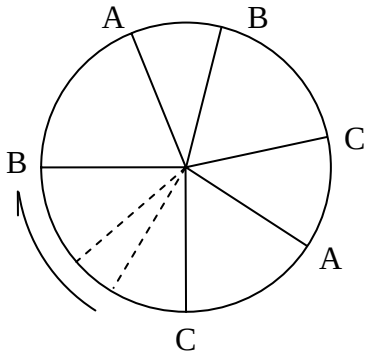

Figure 3.3: Consistent hashing with $n = 3$ and $k = 2$. Both of the two new items (indicated with dashed lines) map to bucket B as it is nearest in a clockwise direction. In case B is removed, only the arcs between (1) southern C and western B and (2) northern A and northern B will have to be re-mapped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

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. Naturally, the entry of the new agent invalidates some of the existing URL mapping. As we have seen, however, the proportion of URLs that have to be transferred from one agent to another is 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 capacity is announced to other agents in a similar manner as the identifier.

UbiCrawler agents traverse the Web 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 DNS resolution results 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 composed 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 scaled by further distribution are hard to incorporate into the model. For instance, an image processing job cannot be handled by an agent different from the one which has downloaded the image in a straightforward manner. In fact, UbiCrawler does not expose a high degree of configurability.

Chapter 4

The component architecture

The future has already arrived. It's just not evenly distributed yet.
William Gibson, American writer of science fiction

Despite the apparently simple basic algorithm of a web crawler, heavy demands in terms of computing capacity and data storage necessitate either very efficient processor and hard disk use, or (as recent research shows) a decomposition of the crawling system spanning across multiple computers. However, coordinating several distributed agents is a much more complex endeavor than managing a centralized architecture due to difficulties in communication and data sharing.

Nevertheless, distributed systems have major advantages over centralized systems [19]:

- *Scalability.* In the case of a distributed system, computing and data storage capacity can adapt to the demands of the job at hand by adjusting the number of crawling agents. In addition, if the agents themselves are scalable, system constraints (most notably available physical memory and CPU share) can often be overcome by allocating more resources to a given agent.

- *Network load dispersion.* Individual crawling agents can be placed at geographical locations traffic-wise near the web domains they are to traverse. This means that data does not have to be migrated to a central crawling agent from a possibly remote location. Consequently, load on network resources is dispersed, or in other words, does not concentrate around a central crawler.

- *Network traffic reduction.* Albeit sending status or control messages to and from a central coordinator may still be necessary, distributed crawlers expose a smaller overall demand on network resources. As agents are geographically close to domains they are to crawl, collecting pages involves a lower total cost. Web site data has to travel less before it reaches a location where it can be processed. Even though status and control messages may still travel larger distances to the coordinator, they are significantly smaller in size.

In order to simultaneously harness the potential in distributed crawlers whilst achieving lower management costs, the author proposes a component-based architecture as a foundation for developing distributed crawlers. Functionality associated with a particular job is wrapped into *components*. Components perform some transformation on their input and produce some output accordingly. For instance, a parser component consumes documents and produces URLs the document contains.

The notion of components is essential because the communication, error recovery and network data transmission aspects associated with a distributed architecture are external to the actual functionality of the component. In fact, it seems straightforward to completely shield the component developer from the intricacy of the distributed architecture. For this end, all components derive from a generic base class that realizes the most common tasks associated with management in a distributed architecture. In addition, the base class handles the most common threading and synchronization issues. Component development may subsequently focus on tasks such as parsing, URL caching, or worker management in a master-worker scenario.

Furthermore, components offer fine-grained scalability. Most distributed crawlers comprise of agents that may be duplicated to cater for scalability. Agents are relatively coarse-grained and are mostly confined to a single computer. Instead, in a component-based architecture, the smallest building block becomes the component. As inter-component coordination is realized in the component base class, components are distributable across computers without difficulty.

In fact, a loosely-coupled component architecture can not only provide inter-component communication services but can also facilitate pluggable components and extension, leading to a simple configuration model.

- Components are *pluggable* if a component can be seamlessly replaced with another that has the same interface. Here, the term *interface* has a different meaning than usual and should refer to the type of inputs and outputs the

component consumes and produces. Two components that have the same interface may consequently have a substantially different behavior.

- An architecture caters for *extension* if it supports integrating newly authored components into the existing framework without possibly major restructuring. As components are independent and virtually defined by their interfaces only, new components are just as easily inserted into the system as components are replaced through pluggability.

In the following sections, the author elaborates a framework that realizes a loosely-coupled component model. The framework will be responsible for component initialization and configuration based on XML documents. Once initialized, it will manage components throughout their lifecycle: possibly remote connections the component requires will be established, data will be automatically marshaled between components and querying the state of each component will be done through the framework.

4.1 Assumptions and 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 class 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

The proposed architecture closely resembles the *pipes and filters* (or chain of responsibility) pattern [23] 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. Filters are *active* in the sense that they drive data flow through the pipeline. Pipes are *passive* building blocks, they are only temporary stores of data and are subject to the discretion of filters.

Nevertheless, the proposed 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 stream of one program is used as an input of the other. In our case, the counterparts of pipes, called *connectors,* are FIFO queues of structured data, i.e. the system realizes an object pipeline pattern.

More importantly, connectors can 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 and produce output. Synchronized, semi-synchronized or unsynchronized scenarios are possible according to whether or not the component fetches all bound inputs from and feeds all bound outputs to the corresponding connectors in a single atomic step. Finally, the proposed system is a pull rather than a push model: components have fair influence in when they consume input or generate output. In the following sections, further details of the system are given.

4.3 Classification of building blocks

The proposed architecture distinguishes three types of building blocks. *Components* and *providers* share some functionality and they encapsulate functional behavior. In contrast, *connectors* are responsible for data transmission.

- *Components* are the primary constituents of the architecture. Broadly speaking, they are responsible for an atomic transformation step. 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. Thus, components consume, transform and produce data. The user-defined functionality they encapsulate is wrapped in an automatically in-

voked method, which performs the actual computation based on input(s) and generates respective output(s).

- *Providers* offer access to external or shared data sources. Providers may be bound to either components or other providers.[1] Each provider acts as a front-end for accessing external data stored in files or databases, or utilizing process-specific resources but providers may also be used in cross-component synchronization. Two major types of providers can be distinguished.
  - *Data providers* marshal data between a component (or provider) and a permanent data storage service such as a relational database. For instance, a unit that executes a stored procedure through a database connection from a pool and possibly returns a resultset is a data provider.
  - *Service providers* offer some process-level service that is not tied to any particular component instance. For instance, a unit that downloads and caches robot exclusion files is service provider. Service providers avoid duplication of memory-resident data structures that are shared by different components. As a result, they can also be utilized in inter-component synchronization.
- *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 fetch 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.

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

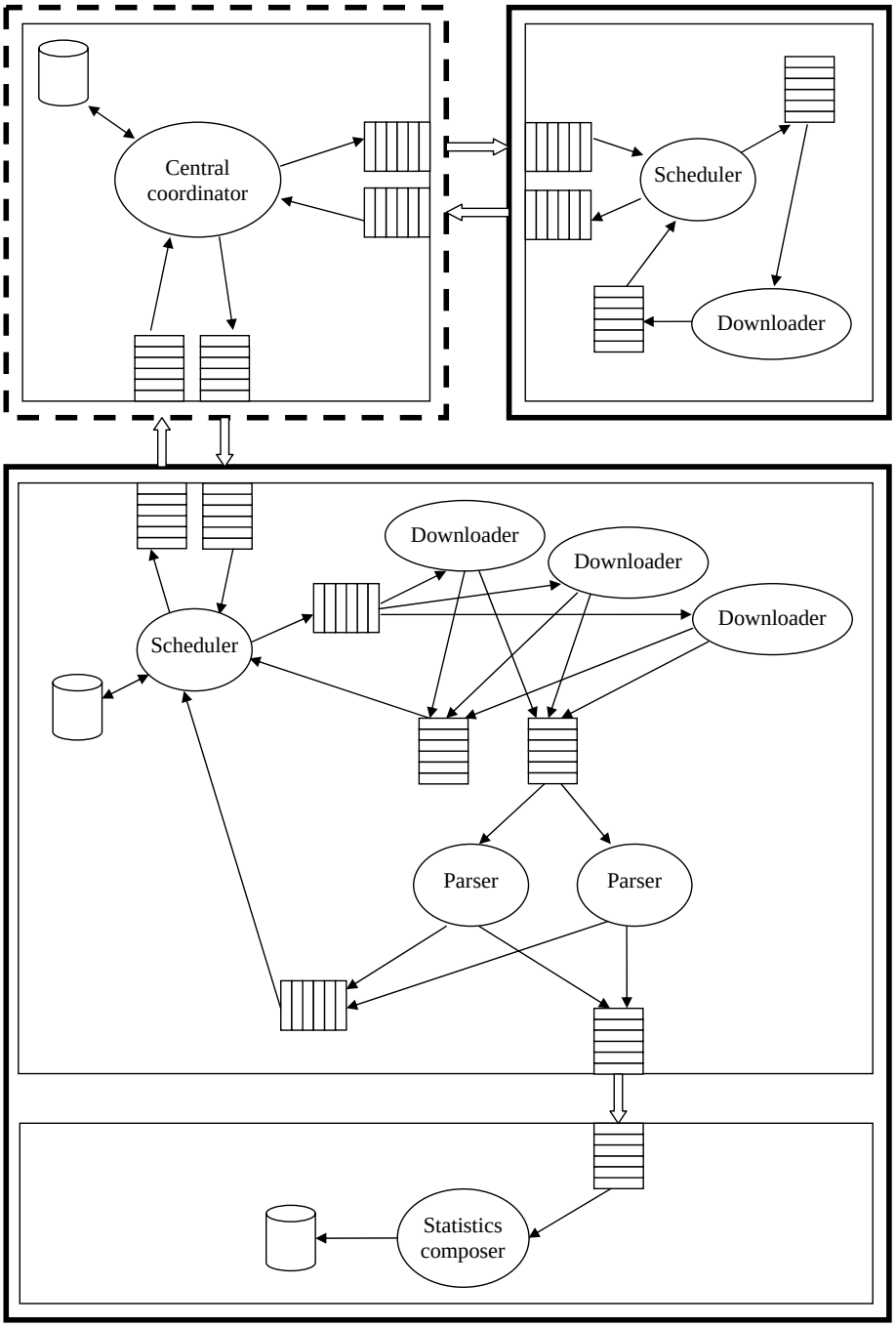

Figure 4.1: A sample component architecture system. Components are depicted as ellipses, connectors as subdivided rectangles (illustrating the underlying buffer) and data providers as cylinders. Process boundaries are shown by thin container rectangles. The boundary of the coordinating master is indicated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

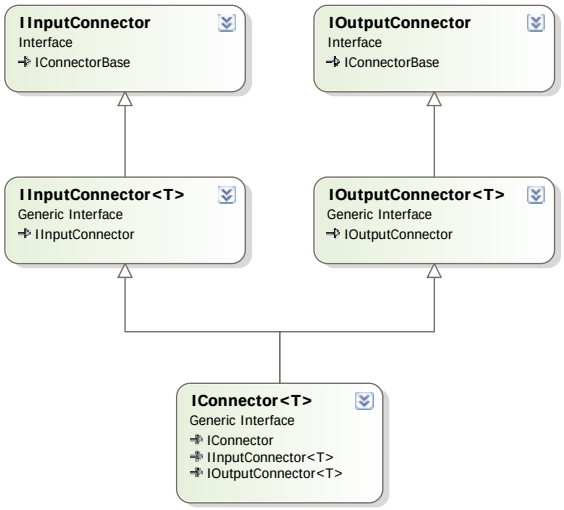

Figure 4.2: Relationship of connector interfaces. The basic connector class in the component architecture directly implements the *IConnector*<*T*> interface.

4.4 Connectors

As briefly mentioned in Section 4.3, connectors are temporary stores of items facilitating data exchange between components. Connectors are *strongly typed,* i.e. they are parameterized with a reference or value type upon startup and accept only elements of this type.[2] 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 even if they adjoin components in the same process. Serializability allows data to be transmitted over the network should the degree of distribution be increased in the system by scattering components across machines.[3]

Even though the most widely spread behavior expected of connectors is acting as a (finite-capacity) FIFO queue, 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 class interfaces a connector should implement (Figure 4.2). Binding components see connectors through these interfaces.

- *IInputConnector*<*T*> is the interface through which consumer components

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

[3]In the current implementation, serialization is performed by means of .NET's built-in binary serializer. For greater flexibility, this could be replaced by XML-based data exchange such as SOAP or a platform-independent communication protocol such as GIOP.

see the connector. It exposes 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 synchronous 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.

- *IInputConnector* and *IOutputConnector* are the weakly-typed equivalents of *IInputConnector*<*T*> and *IOutputConnector*<*T*>. They are intended for cases when connectors have to be uniformly treated regardless of the parameterizing type e.g. when storing them in a typed collection such as a *List*<*T*>. Nevertheless, placing an object of improper type in a connector should raise an exception.

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 connector on the site A; and (b) a receiver component and a local connector bound to the receiver on site B (Figure 4.3).[4] 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

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

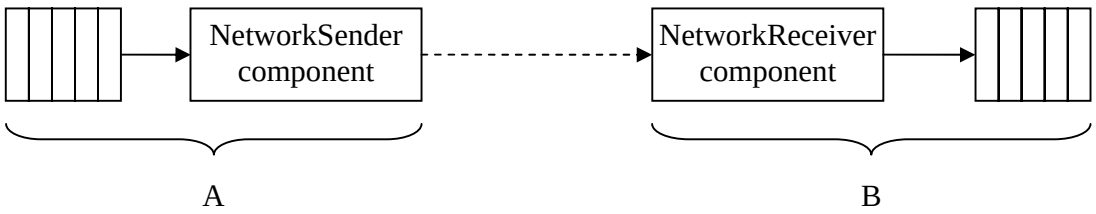

Figure 4.3: The internal structure of a uni-directional remote connector. The continuous line indicates pass by reference, the dashed line corresponds to serialized data transmission through the network.

connector, no indication is ever sent to the producer component when the transmission was successful.[5] 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 unless the asynchronous connector data exchange methods have been used.

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 attempt feeding items to the connector.

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

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 exchanged.

4.5 Providers

In the component architecture, providers play a fundamental role in accessing external data sources (*data providers*) and using resources shared by multiple components (*service providers*). A provider is a regular class that exposes public properties and methods, which can be get, set or invoked by components (or providers) that bind to the provider. However, the component (or the other provider) sees the provider through a class interface and is unaware of the actual implementation behind that interface. This caters for pluggability: the provider implementation can easily be replaced by another without affecting the structure of the system.

For instance, a URL filter component could call a method of a URL seen provider to check if a URL has previously been referenced. The provider class interface seen by the component exposes an *IsUrlSeen()* method that performs the test. The actual implementation of the provider queries an underlying database to see if the URL exists. Nonetheless, implementation details are invisible to the component. Indeed, if the implementation is replaced by a probabilistic structure, such as a Bloom filter [13], changes are required only in the system configuration file to use a Bloom filter instead of a database-backed implementation. In fact, by changing the provider class the component binds to, the system will run without recompilation.

The ability of replacing the provider implementation behind class interfaces is a useful merit of the component architecture. By means of this technique, transparent caching mechanisms can be incorporated into the system. Suppose that component C binds to the data provider D through the provider class interface P. D is backed by a file on the hard disk and can at times be very slow. However, a buffering mechanism to cache most recent results could significantly improve speed. Indeed, inserting a new provider B between C and D is by no means a difficult task. Provided that B binds to D through the original interface P while exposing the same interface toward C, C will operate seamlessly.

The provider binding model through well-defined interfaces is very similar to the object-oriented notion motivating the use of interfaces. However, the model used in the proposed architecture is a late-binding model, whereas in the object-oriented world, early-binding is more common. In fact, it is not until initialization that the

actual participants in a binding are instantiated and coupled. In the classical model, coupling is performed compile-time.

Proper synchronization is essential when developing providers. There is no guarantee that a provider property or method is accessed by a single component only at a time. As each component may execute on a separate thread, providers should employ mutual exclusion devices to prevent data corruption.

Components and providers share some operations. In particular, both have unique identifiers, settings and performance metrics, and both may bind to (other) providers. In order to avoid duplication of common functionality, both components and providers derive from the abstract base class *GenericBase*. The methods of this base class are elaborated in Section 4.6

4.6 Components

Components constitute the core of the architecture. Although they may realize a large variety of different functionality, it is of paramount importance that the framework implement basic initialization, communication and configuration services that are common to all components. In particular, it should facilitate component-to-connector and component-to-provider binding, configuration via XML files, querying configuration settings and performance metrics at run-time, and suspension and restart on need. Hence, in the proposed framework, all components derive from *GenericComponent*, an abstract class which acts as the root of the component hierarchy. *GenericComponent* exposes the following methods:

- The *BindProvider()* method is inherited from *GenericBase* and helps bind the component to providers. This method is invoked by the framework at run-time during system initialization.

- The *BindConnector()* family of methods aid binding input and output connectors to the component. They interpret component annotation (see Section 4.6.1) and perform type validation. These methods are invoked by the framework to configure the component on startup after providers have been bound.

- *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

and can be used to allocate resources that assume the presence of bound connectors and providers. (See Section 4.8 for defined thread types in the framework.)

- The framework launches the component by means of its *Start()* method. The *Start()* method is assumed to return, possibly by initiating an asynchronous but not necessarily finite operation.

  In this architecture, components are active units in the sense that they drive data flow through the application. This means that each component should be associated with a thread that performs not only data transformation but also interaction with its environment, consuming and producing data, and calling provider methods, in particular. The framework provides the *GenericThreadedComponent* class to develop threaded components.

  Most components realize a *single-threaded model*. Here, a dedicated thread is associated with the component. It extracts data from input connectors, transforms it, and feeds the result of the transformation into output connectors. A single-threaded model eases component development because no mutual exclusion devices are required on data structures.

  Nonetheless, components may use the *worker-thread* or even more sophisticated multi-threading models. Multiple threads should be registered calling the *RegisterThread()* method so that proper cleanup is ensured once the component is destroyed. *Start()* is inherently more complex for these component types. However, creating components with such intricate threading models is seldom necessary. Instantiating multiple components of the same type is often an easier way to achieve the desired effect.

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

- Enumerator methods return providers, input, output or all connectors bound to the component. They are used by inheritors of *GenericComponent* to perform connector bindings based on component annotation. (See Section 4.6.1.)

- *Dispose()* releases all resources claimed by the component during initialization, including any executing threads.

4.6.1 Component annotation

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

The component architecture features 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 startup 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 couples a data or service provider rather than a connector with the component. Similarly, it takes a role and a type parameter. In this case, however, the type parameter identifies the type of the expected provider class or interface.

Component annotation helps delay component, connector and provider coupling until run-time initialization. The meta-information associated with components defines the expected interfaces but does not specify the actual behavior or implementation. It is the configuration file that tells the framework what kind of classes to instantiate and how to interconnect them to make the system operable.

4.6.2 Component types

Components are usually not written from scratch by directly inheriting from *GenericComponent*, the component base class. One can easily identify certain consumer

and producer patterns and generalize them (Figure 4.4). In the following sections, general component types the framework supports will be introduced.[6]

4.6.2.1 SimpleConsumer

A *SimpleConsumer* is a special component that produces no output yet it consumes input. *SimpleConsumer*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 as they require no further processing.

4.6.2.2 SimpleProducer

A *SimpleProducer* generates output data but consumes no input. They are handy in generating test input or signals based on state change. Cooperating with a provider, *SimpleProducer*s can generate output when a condition on a shared data structure is satisfied. For instance, a *SimpleProducer* might feed URLs into the system that are fetched from a database through a provider whenever new items are available.

In addition to the aforementioned scenarios, *remote connectors* are also realized with *SimpleProducer*s, *SimpleConsumer*s and regular connectors. The sender, which is a *SimpleConsumer* instance, consumes items from a local connector and transmits them over the network. On the remote site, the receiver, which is a *SimpleProducer* instance, feeds items back into another connector once it has received transmitted data (Figure 4.3).

4.6.2.3 SimpleFilter

A *SimpleFilter* is a regular component that consumes items from a single input connector and produces zero or more items corresponding to each input item. *SimpleFilter* derivatives should override the *Filter()* method. When data is available in the input queue, the framework automatically invokes the method, which is to return a list of output items. The resultant items are appended to the output connector once free slots are available. When all output items have been fed into the

[6] 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.

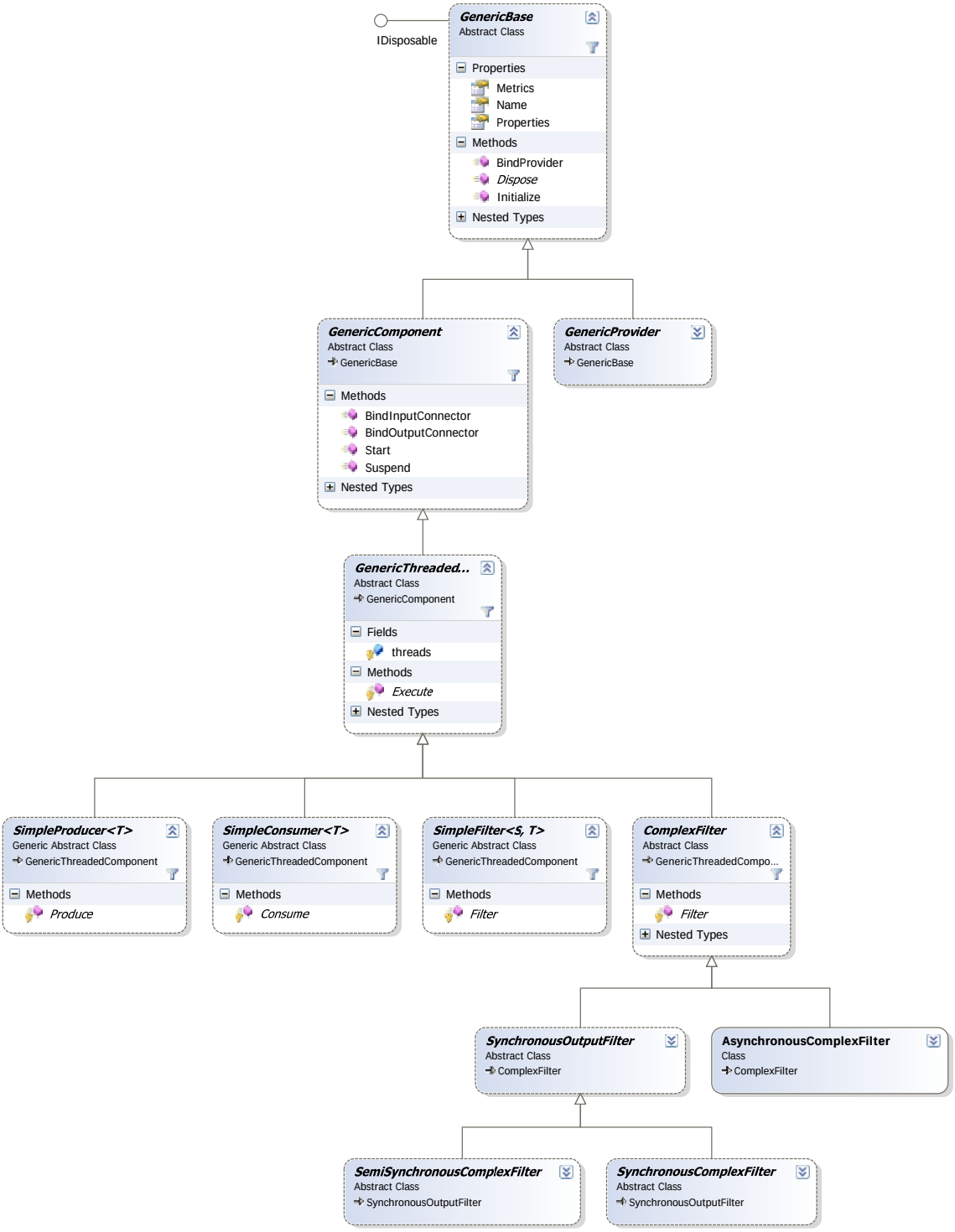

Figure 4.4: The hierarchy of classes comprising the component architecture. Component implementors should derive from one of the descendant classes rather than directly from the uppermost superclass.

respective connector, the process is repeated. The *Filter()* method has the following signature (S signifies the input (source) data type, while T denotes the output (target) data type):

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

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 4.5).

```
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 4.5: A minimalistic example *SimpleFilter*. The types the component consumes and produces are encoded into template parameters.

4.6.2.4 SynchronousComplexFilter

A *SynchronousComplexFilter* is a more powerful *SimpleFilter* variant in the sense that it can consume input from and produce output to multiple connectors. However, as the name suggests, it operates in a synchronous manner, 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.[7] In Figure 4.6, the *SimpleFilter* example shown earlier is rewritten as a *SynchronousComplexFilter*.

In this case, the architectural pattern is more general than in Figure 4.5 as it should be prepared to accept multiple input and output connectors even if not the case in this particular scenario. Hence, item 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 optional in the *SimpleFilter* case. If specifying a single input and/or 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. Roles are ideal in expressing the relationship between bound connector and binding component.

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

[7] 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 [3] does. Instead, we refer to both concepts unanimously as *property* and explicitly state if fields are not allowed.

4.6.2.5 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. After a single input property has been set, all output property values or collections are inspected by the framework if they are non-default[8] (for scalars) or contain any elements (for collections). If so, they are fed synchronously into output connectors. (Figure 4.7)

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

4.6.2.6 AsynchronousComplexFilter

An *AsynchronousComplexFilter*, while resembles a *SynchronousComplexFilter*, is a much more complex component. It is fully asynchronous, i.e. there is no respective

[8]The default value for reference types is the null reference. For value types, it depends on the (comprising) primitive types, such as 0 for integers or 0.0 for doubles.

order in which input and output properties are get and set. Whenever one or more items are available in any of the input connectors, the corresponding input properties are set in no particular order. Similarly, whenever free slots are available in any of the output connectors, output properties are queried.

*AsynchronousComplexFilter*s rely on a worker-thread mechanism. A set of workers is available at the component's disposal. Whenever pending data is available in any of the bound input connectors, an identifier token is deposited in a job queue. Similarly, whenever free slots are available an output connector and the component is ready to supply corresponding data, a token is deposited. Idle worker threads fetch tokens from the deposit and perform the associated job.

4.7 Configuring and monitoring units

Configuration in the component architecture concerns two distinct aspects. *Unit interconnection* defines the topology of components, connectors and providers that run in the same process. In contrast, *configuration settings* fine-tune the behavior of each component and provider by setting property values.

Initial unit interconnection is defined by means of a process-specific XML file. Not only does it specify the exact components and providers that should be instantiated, it declares which component binds to which providers and connectors by means of unique identifiers and roles. Recall that component annotation only defines the expected interface of a provider or connector. In the case of providers, this means the specification of a provider interface; for connectors, the type of items to be consumed from or produced to the connector, depending on its function. The exact class behind the interface is not visible to the binding component but is declared in the XML configuration file.

Apart from declarative configuration by means of XML files, the architecture supports on-the-fly reconfiguration. Components expose 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. Such properties are wrapped in a *PropertySheet* class that is specific to each component although they have a common superclass. The framework supports retrieving a *PropertySheet* object for each component. The actual properties of a *PropertySheet* object may be discovered by reflection and displayed on a user interface. The modified object may travel back to the component, which may subsequently update its internal properties based on the values in the *PropertySheet* object.

Additionally, connector and provider bindings may also be changed run-time. As both connectors and providers are seen through interfaces, switching the underlying object does not generally disturb behavior. For instance, a DNS resolver that uses a binary tree data structure can easily be replaced by one that uses a hash table while the system is running.

However, in order to avoid data corruption, components must be suspended prior to setting new configuration values by means of the *Suspend()* method. 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 should be called for each of these before they are bound.

Monitoring components and providers is done in a similar fashion to dynamic configuration. Performance metrics are wrapped in a *Metrics* object, which is specific to each component and provider albeit they have a common base class. This object is forwarded to the requesting party on demand, which is most often a graphical user interface. Again, the actual properties in the object are discovered through reflection. Contrary to configuration settings, however, performance metrics do not round-trip: it is meaningless to return such objects to components which may recompute metrics themselves.

4.8 Threading

So far we have seen that the architecture comprises of independent components in terms of data encapsulation. Each component is defined by its interrelation to other units and has its own configuration settings. However, components are also independent entities computation-wise. Each component is associated with at least a single worker thread that is responsible for consuming input and/or producing output. Action methods, such as *Filter()* in *SimpleFilter* are executed periodically by this worker thread. Below threads that exist in the system are elaborated in more detail.

The *initializer thread* is the main thread of the application when the framework process starts. Its primary task is to set up components, connectors and providers based on XML configuration files and initiate component threads on startup. In particular, component *Initialize()* and *Start()* methods are invoked from the initializer thread. After initialization is finished, this thread terminates.

Component threads are associated with each component and manage objects and resources held by the component while it is running (i.e. it is not in a suspended state). For synchronous components, a single component thread marshals input consumption and output production. In contrast, asynchronous components use the worker-thread model; separate *component-local worker threads* perform this task. Recall that components are active units in the architecture, whereas connectors and providers are passive units. Therefore, providers and connectors have no associated threads, their methods are called by components.[9]

Global worker threads live in a thread pool managed internally by the .NET framework awaiting for asynchronous jobs to be executed. For instance, long-lasting

[9]As remote connectors comprise a local connector and a network sender or receiver, they have a single consumer or producer component thread, depending on the direction of data flow.

operations such as a file transfer or a DNS resolution can use a worker thread to be notified of completion instead of synchronously waiting for a possibly lengthy operation to finish. Global worker threads help avoid thread initialization overhead of asynchronous operations by keeping threads alive even if they have completed the job assigned to them.

Besides asynchronous jobs requested by components, the component framework uses global worker threads to handle queries for configuration settings and performance metrics through .NET remoting services. External calls to methods of marshal by reference objects are executed on the worker pool.

4.9 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 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 (or parser). The extractor reads documents from its input connector and produces URLs to its output connector. The two components are basically independent. However, in some situations, components have to cooperate. Cooperation undoubtedly requires some form of coordination between the participating components. Nonetheless, we wish to avoid strong coupling between the two. Two possible ways exist to address this matter:

The first solution, *provider-based cooperation* can be classified into local and remote cases (Figure 4.8).

- 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.

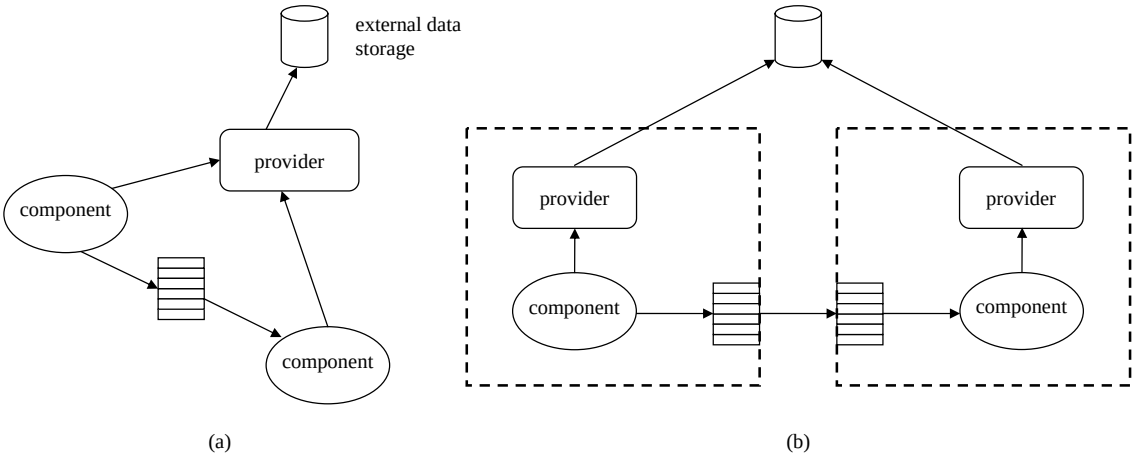

Figure 4.8: Provider-based cooperation. (a) Local cooperation. (b) Remote cooperation. Dashed lines indicate process boundary.

Clearly, for maximum scalability, remote provider-based cooperation should be used whenever possible. Whenever different types of components cooperate, remote cooperation shows the same behavior as local cooperation but the former benefits from fewer constraints at a small speed penalty. On the other hand, local provider-based cooperation has its uses. In particular, it is ideal for sharing 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. This is especially true for process-level resources or services. For instance, multiple downloader components may be used to increase download rate but only one DNS resolver is required for the whole process. An IP address cached by the resolver during a request by one downloader can be re-used by another downloader.

In *connector-based cooperation*, connectors serve as message channels between components. In this model, components are interconnected by connectors in both directions so that a component may respond to the data it receives via acknowledgment or other statistical messages. In particular, connector-based cooperation can be used to send informational messages about the completion of an assigned job. For instance, a downloader component may signal downloaded documents to the host-specific selection strategy by emitting informational elements into a connector. This cooperation strategy does not distinguish between local and remote cases. The difference is only in terms of speed; remote connectors involve a greater delay.

Chapter 5

The crawler application

Be fruitful and increase in number;
multiply on the earth and increase upon it.
Genesis 9:7

In Chapter 4, we have seen how a component architecture can ease the development of distributed crawlers. Notably, the base classes of the architecture support asynchronous, message-based, inter-component communication and data access to process-level, possibly external shared resources. As a result, low-level issues that are required by all distributed components are encapsulated by generic component and provider classes. Inheritors need not re-implement these services but they can implicitly re-use them simply by overriding member functions and using the annotation mechanism.

In order to demonstrate the feasibility of the architecture discussed in the previous chapter, the author describes a distributed web crawler that features a central coordinator. The coordinator, subsequently referred to as the *master,* is responsible for partitioning the web into disjoint sets of domains. Besides, its secondary objective is assigning domain sets to *workers.* As their name suggests, workers are the essence the system: they are responsible for traversing the domains they have been assigned, that is, they perform the actual crawl.

As the distributed crawler makes full use of the previously introduced component architecture, the notions of master and worker do not explicitly refer to computers. In fact, neither is confined to a single machine, or alternatively, the entire system may run on a single machine. In other words, the division into roles master and worker is orthogonal to the across-machine allocation of components that comprise master and worker.

5.1 Degree of distribution

It may give rise to disambiguity that the author refers to a *distributed* crawler with a *central* coordinator. Nonetheless, the simultaneous use of both concepts is not a contradiction. Albeit extremities exist, the terms *distributed* and *centralized* express a measure of the degree of distribution rather than referring to boolean values.

In fact, the proposed crawler is distributed in the sense that its functionality is not concentrated in a single machine that communicates with the rest of the world like in the case of the Mercator [25] crawler, presented earlier as related work. In contrast, independently executing, cooperating components make up the system, each of which may run on a separate machine. However, the system is centralized in the sense that there is a single point in the system which is a source of information that does not span across machines. Clearly, workers are unaware of the state of their companions, they have no information whatsoever concerning the reference structure or crawl status of domains they are not responsible for. On the contrary, the master possesses a complete knowledge of the domain-to-worker assignment at any point in time, even though it neither has any clue about the internal state of workers. Summarizing, the system is fully distributed with the exception of domain-to-worker assignment, which is centralized in the master.

Systems exist that regard this centralization of information a serious drawback and strive to avoid any single points of failure. For instance, UbiCrawler [14] shows strict fault-tolerant properties. However, the author believes that tolerance of permanent faults is not a critical issue in web crawling. Worker processes running the component architecture survive if the master component is temporarily unavailable. In fact, items start accumulating in remote connector buffers until they are ready to be sent to a restarted master process. If the buffers become full, no new items are accepted and the producer component is halted. This behavior propagates in the system until proper operation is restored. The same argumentation is valid in the case of failing workers. However, as the master must never suspend operation as it cooperates with many workers at a time, data has to be written to disk when buffers are full rather than pausing until free slots are available. Meanwhile, the possibility of configuring the web partition strategy at a central location leads to much easier manageability than if this information would be assembled individually by workers, as in UbiCrawler.

5.2 Worker components

Workers are the major unit of replication in the proposed system, each of which connects to the master. They are responsible for the traversal of domains they have been assigned and maintain related data structures. In particular, each worker has its own URL queue and URL-seen structure as seen in the crawler algorithm (Figure 1.1).

A minimal worker configuration and the data flow between its components is seen in Figure 5.1. The roles of the particular components are briefly discussed in the following paragraphs. Nevertheless, a detailed description of how these components work is beyond the scope of this thesis. The aim is to give a basic idea of how the crawling process is realized.

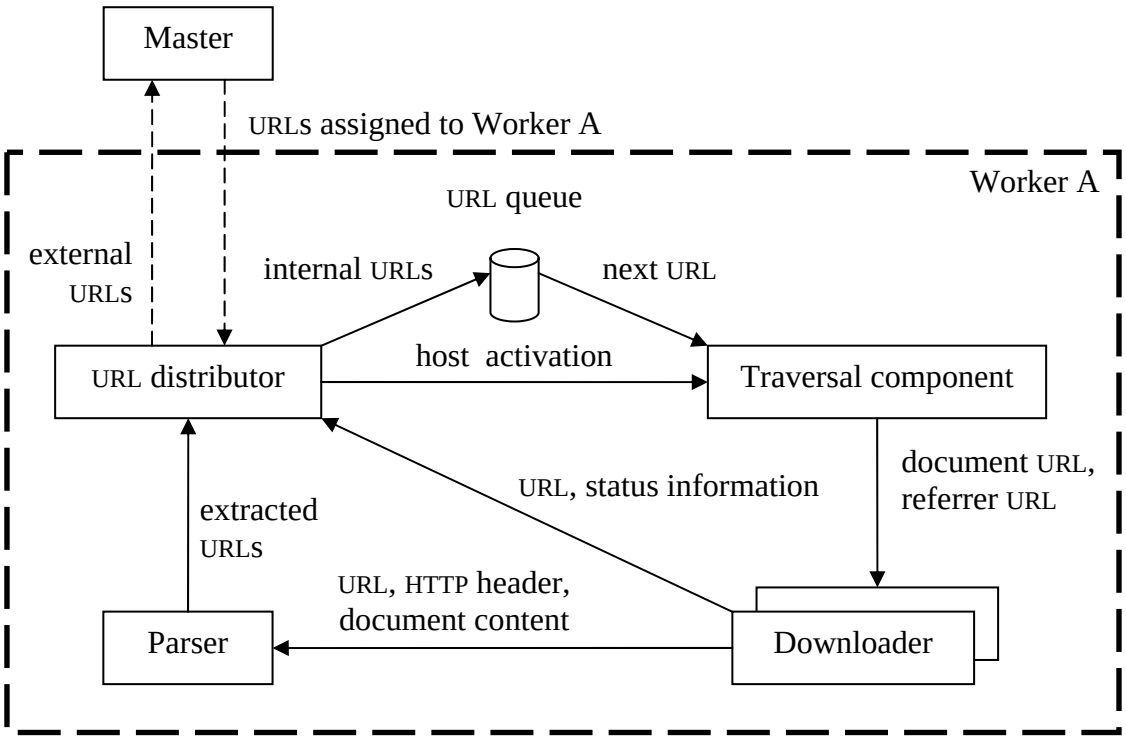

Figure 5.1: A system configuration that provides minimum functionality. The rectangles represent the components and the arrows between them indicate the direction of data flow.

Scheduling URLs to documents awaiting to be downloaded are stored in a URL queue on each worker. The URL queue is a disk-resident data structure. During scheduling, URLs in the queue are fed to downloaders in a respective *order* with

proper *timing*. Thus, scheduling is split into two aspects: the order of documents is determined by the *traversal component*, while timing is ensured by the *load balancer component*.

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 a mere 10%), which would generate high demand for a single host and run the risk of overloading it. 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.

Load balancer component 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 used.

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.

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

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

5.3 Master components

As previously noted, master components coordinate workers by receiving external URLs a given worker is not expected to process and forwarding these URLs to either (1) the respective worker already in charge of the domain or host the URL refers to or (2) a newly assigned worker 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 directly provides static interconnection only, i.e. the addressee is independent of the transmitted content.

In order to tackle content-dependent item designation, the master 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 bidirectional remote connector from the perspective of workers in the sense that it acts as the master-side stub of the worker-side remote connector. The items the communicator component receives are fed into a common inbound pool connector from where they are extracted by the *marshaler component* that assigns a worker to each item it receives. In the assignment process, each item is labeled with a worker identifier and the tagged items are placed in an outbound pool connector. The communicator fetches items from this latter connector and forwards them to workers based on the identifier.

5.3.1 The marshaler component

Marshaler components form part of the master and are the primary agents of worker coordination. They are single-input-connector, single-output-connector components (instances of *SimpleFilter*) with three providers. Marshalers receive URLs and *designate* URLs to workers. URL designation is a multi-step process.

1. The marshaler component fetches a URL from its input connector.
2. The URL is tested against a *recently seen table*.
3. The URL is canonicalized.
4. URLs violating domain constraints are discarded.
5. The host part of the URL is generalized to an *assignable domain*.
6. The assignable domain is designated to a worker.

Let us investigate these steps in more detail.

Step 1: Fetching a URL In this first step, the syntax of the URL extracted from the input connector of the marshaller component is verified. Ill-formed URLs, such that those with an invalid URI scheme are automatically discarded. For instance, URIs with the scheme *mailto* indicate e-mail addresses and hence cannot be traversed. It is meaningless to designate such URIs to workers.

Step 2: Testing if URL is recently seen URL testing is performed 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 worker. 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 [17].

Clearly, a large number of web pages link to a small subset of the entire web. In fact, the recently seen table can be very effective because URLs to be marshaled follow a Zipfian distribution. Zipf's law can be formulated as:

$$f_{s,N}(k) = \frac{\frac{1}{k^s}}{\sum_{n=1}^{N} \frac{1}{n^s}} \tag{5.1}$$

In Equation 5.1, N is the number of elements, k is an integer in the range $[1; N]$ denoting their rank, and s is the exponent characterizing the distribution (often letting $s = 1$). In other words, $f_{s,N}(k)$ will be the fraction of the time the kth most common web page is referenced by a URL. As a result, the recently seen table can filter out commonly referenced URLs without forwarding it to workers for processing, thereby saving considerable bandwidth. A table of 10 000 to 100 000 URLs can significantly reduce communication overhead [19].

Step 3: URL canonicalization In this step, the host name part of URL is canonicalized and IP addresses are resolved into host names (inverse resolution).

A host can have multiple names yet a single name is *canonical* and others are referred to as *aliases* [28]. 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.

Step 4: URL constraint validation In order to enforce traversal constraints, URLs with host names that do not satisfy domain constraints are discarded. Section 5.3.2 elaborates on how constraints are set and verified.

Step 5: URL host name generalization Before a URL can be assigned to a worker its host name must be generalized to an *assignable domain*. An assignable domain usually corresponds to a secondary domain, such as `hu.bme`.[1] However, in certain situations, the secondary domain would cover too large a portion of the web to be traversed by a single worker. Such an example is the `uk.co` domain, which constitutes an "umbrella" domain rather than one that belongs to a particular organization. In these latter cases, `uk.co.example` is assigned rather than `uk.co` itself.

Step 6: URL designation In the final step, it is verified if the assignable domain is already mapped to a worker. If not, the unassigned domain is mapped to a worker based on the assignment algorithm. Both verification and assignment are database-backed operations implemented in a provider which returns the globally unique identifier of the worker if a mapping exists, or a newly assigned worker if not. The assignment algorithm can range from simple random assignment to more sophisticated geographical or domain-based assignment. Again, the algorithm is realized by means of a provider.

Once the responsible worker is identified, the URI is extended with the assigned worker identifier and the resultant *designated* URL is placed into the output connector. Designated URLs are instances of the generic class *DesignatedItem*<*T*>, which

[1] In this chapter, we use the reverse notation [28] for domain names as it more appropriately reflects the hierarchy in URLs. Each URL starts with the most general domain and ends with the most specific one. Hence, `www.example.com` is written as `com.example.www`. Nevertheless, the presence of the root domain is not indicated with a leading dot.

associates an item of type T it encapsulates with a unique identifier. In this scenario, the identifier corresponds to a worker identifier and the encapsulated item to a URL.

5.3.2 Domain constraints

Domain constraints are an extremely simplified variant of regular expressions [22] that aim to test if a host name (or domain) is a descendant of another domain, the latter of which represents the constraint. Just as domain names, constraints are hierarchical where parts are separated by dots. For instance, the `org.wikipedia.en` host is a direct descendant (subdomain in DNS terminology) of the `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, the constraint `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, the constraint `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 worker for processing. For instance, the host name `org.wikipedia.en` does not match the affirmative constraint `hu.*` and is discarded in a designation process. 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 (top-level) domain. A crawler that collects only pages in the `hu` domain has an affirmative domain constraint `hu.*` in effect without any negative constraints. Obviously, affirmative and negative constraints cannot overlap, otherwise negative constraints take precedence.

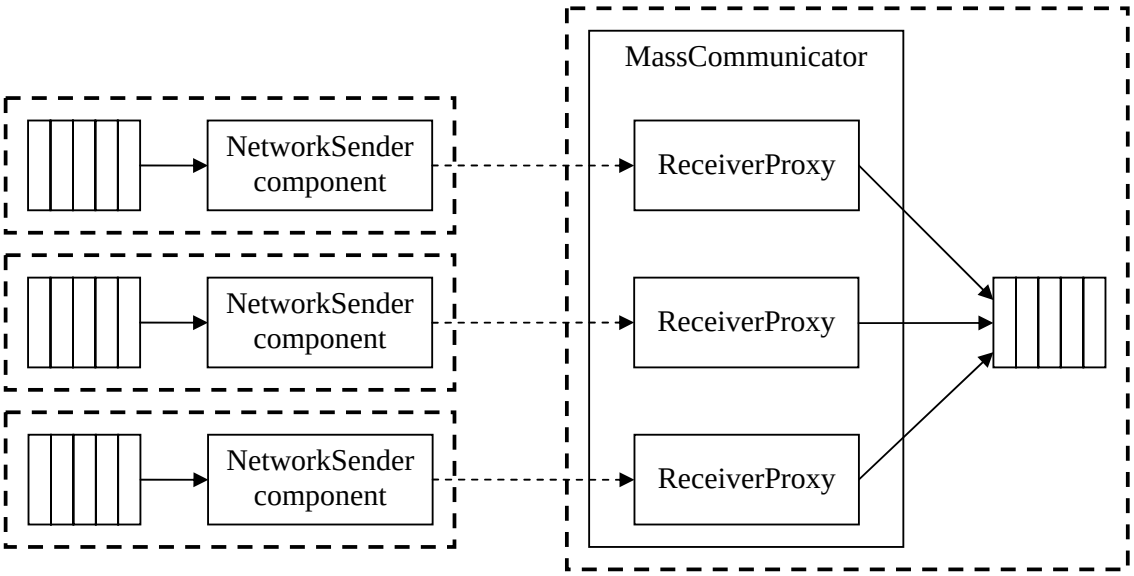

Figure 5.2: The mass communicator component is seen from the worker-side as the master-side constituent of a remote connector. Dashed rectangles indicate process boundary and dashed arrows correspond to serialized communication. (Compare to Figure 4.3)

5.3.3 The mass communicator component

The mass communicator component simultaneously acts as the master-side stub of multiple worker-side remote connectors. For each worker, it instantiates a *ReceiverProxy* and a *SenderProxy* object, both of which represent a connection to the remote worker. When a worker-side remote connector transmits a data packet to the master, the *ReceiverProxy* object intercepts the packet, extracts items it contains, tags them with the worker 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 and transfers them to the respective worker.

The communicator component is resilient against temporary failures. Whenever the connection to a worker 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 as if they were newly arriving URLs.

5.4 The graphical user interface

In monitoring each worker process and hence the overall performance of the system, a graphical user interface (abbreviated as GUI) is an indispensable tool. The primary role of the user interface is to visualize the topology of a worker process, i.e. how components and providers are interconnected by means of connectors. Additionally, the GUI displays detailed information on each component, provider or connector, such as its identifier, miscellaneous configuration settings and performance metrics. Figure 5.3 shows the different parts the graphical user interface is made up of. The exact semantics of each user interface element is elaborated below.

- The *site browser* contains *Site* objects, which identify the name, description and location of each worker process the GUI is aware of. A *name* is a human-readable short identifier to be displayed in the drop-down list. A *description* is an optional longer piece of text that adds auxiliary information. A *location* is a compulsory parameter for each worker and comprises of the following constituents:

  - A communication *schema* such as HTTP or TCP tells the user interface which protocol to use to connect to the worker process. The schema must match the protocol specified in the worker process configuration file.
  - A *host address* specifies the DNS name or IP address of the host to connect to. This is a dynamic property specific to the computer on which the worker process is running.
  - A *port* defines the port to bind to. This is configured in worker configuration files.
  - As multiple worker process services may be connected to the same port on the remote machine, a *path* is used to distinguish between these services. The string `Topology` should be used to fetch topology information.

  Sites are loaded on startup and saved on termination to a GUI-specific XML configuration file. In addition, the list of sites is editable run-time.

- The *topology visualizer* depicts the way components, providers and connectors are interconnected in the worker process selected in the site browser. Coloring helps distinguish different units. As the interconnection may correspond to a complex graph, a force-based layout algorithm is used to arrange

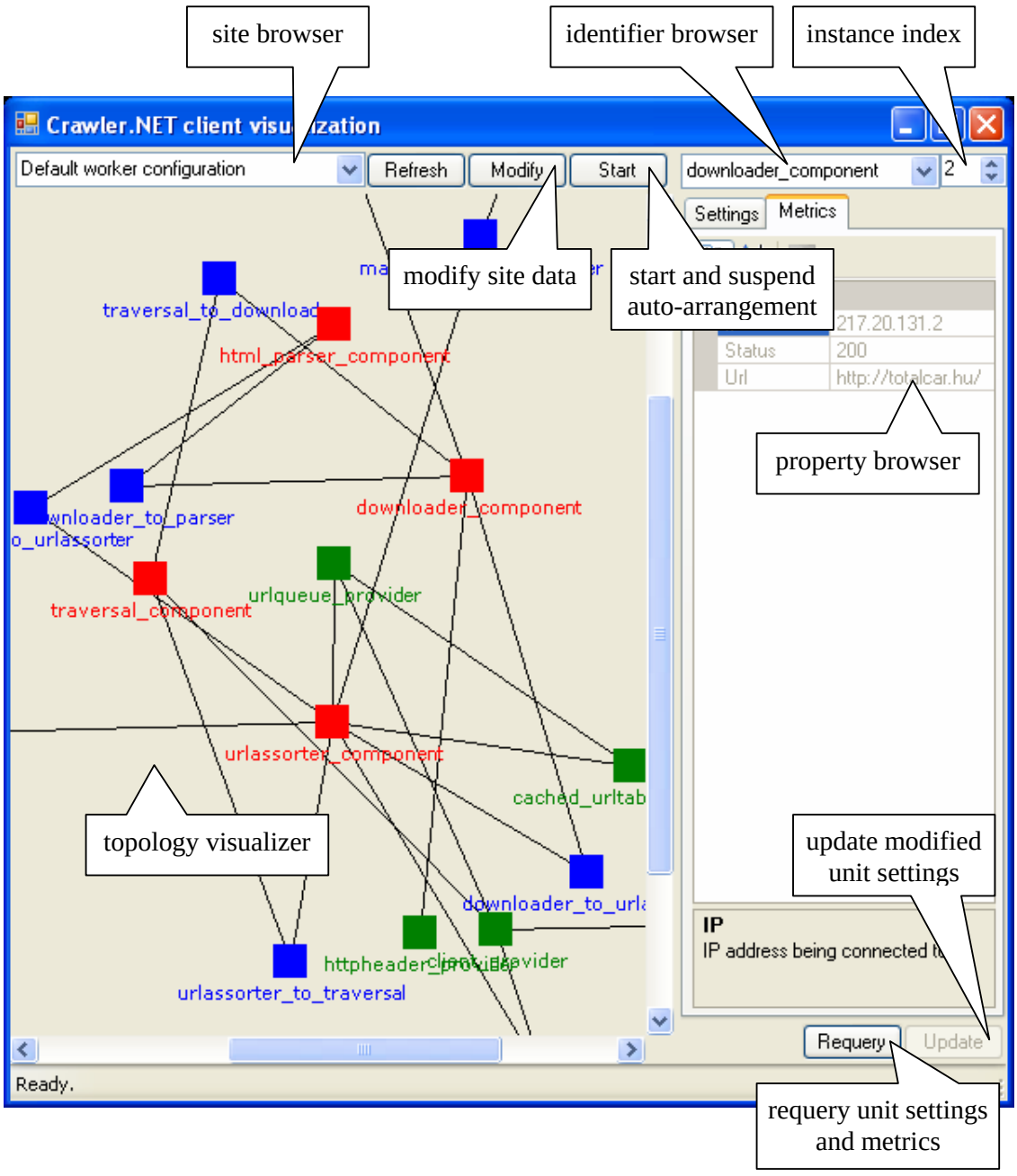

Figure 5.3: Elements of the graphical user interface.

nodes. Despite their simplicity, force-based layouts offer good quality results for medium-size graphs in terms of such aesthetic properties as uniform edge length, uniform vertex distribution and showing symmetry [4]. The topology visualizer implements a force-based layout algorithm using Hooke's law and Coulomb's law. Auto-arrangement is performed interactively, and can be started and suspended on need. Additionally, the visualizer allows direct user intervention: manual rearrangement helps remedy the most serious drawback of the layout algorithm that it may get stuck in local minima.

- Units are associated with a unique *identifier* in the worker process configuration file. To quickly locate any unit in the topology panel, the identifier is displayed as a label below the graphical representation of each. However, a unit can also be selected using the *identifier browser*.

- Multiple instances of the same unit are shown as a single node in the topology graph. In order to distinguish between instances, one must make use of the *instance index browser*.

- The *property browser* exposes the configuration settings and performance metrics associated with each unit. Configuration settings are both readable and writable properties whereas metrics are read-only. Current settings are retrieved by an explicit query or they are fetched automatically when the topology is displayed the first time or is refreshed. Similarly, any changes made to settings take effect when explicitly requested. In contrast, metrics are updated periodically without any user interaction.

From a design perspective, the graphical user interface is strictly detached from the individual worker processes. Both configuration settings and performance metrics are retrieved using .NET remoting services. The GUI instantiates a stub for each worker process that it monitors. The stub forwards method calls to the remote worker process and marshals the reply it generates to the GUI. In order to minimize network latency, both configuration settings and performance metrics are retrieved in bulk whenever possible, which is of special importance in the case of the latter, which are frequently updated.

The user interface is a "black-box implementation" in the sense that it treats components, providers and connectors opaquely, it does not exploit any special characteristics of these units. The topology of a worker process is transmitted to the GUI as an annotated graph, which comprises a set of named nodes and connecting

edges. Configuration settings and performance metrics are encapsulated in *PropertySheet* and *Metrics* objects, both of which are attached to named nodes. The property browser uses reflection to discover the actual properties in *PropertySheet* and *Metrics* instances and display appropriate editor interface [3].[2]

[2]The annotated graph representation and the base *PropertySheet* and *Metrics* classes are defined in a core assembly the GUI directly links to. However, actual *PropertySheet* and *Metrics* inheritors are auto-loaded by the user interface run-time, which means that they should be accessible in an assembly co-located with the GUI assembly so that they can be instantiated.

Chapter 6

Evaluation

We can only see a short distance ahead,
but we can see plenty there that needs to be done.
Alan Turing

By having introduced a distributed component architecture in Chapter 4 and by creating a crawler reference implementation based on the architecture in Chapter 5, an extensible distributed crawling system which possesses both the features scalability and easy management has been realized. In this concluding chapter, the author compares the proposed crawler to related work, shows some dynamic properties of the system, summarizes research and lays out perspectives for future work.

6.1 Comparison to related work

Table 6.1 assesses the proposed system to related work described 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.

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 platform	single machine	LAN	not restricted	
Scalability	limited		not limited	
Primary limitation	disk speed	inter-comp. communication	suboptimal traversal	n.a.
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.

- *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.

- *Target platform* 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.

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

- 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, "self-healing" operation in the case of permanent errors.

6.2 Dynamic properties

Although the scalability of the outlined architecture relaxes the strain on computing resources, communication delay and the relative overhead of the 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 ran 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 connec-

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

tors 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 [33] 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 the proposed 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 proposed architecture 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 describing how individual components are spread across machines. Thus, the system is loosely coupled, easily configurable and extensible.

In order to prove that the architecture introduced is sound, an own distributed web crawler has been implemented. It is a coordinated crawler consisting of workers and a single master (or possibly more cooperating masters). The distinction between master and workers is only conceptual, both the master and workers themselves consist of multiple components and are therefore distributable across several machines.

The proposed crawler realizes a dynamic partitioning of the web by the master assigning domains to workers. The master comprises of communicator components that maintains a network connection to workers, and a marshaler component that designates domains to workers. Workers crawl the domain they have been assigned and transmit foreign URLs back to the master for re-assignment. The system achieves zero overlap, maximum possible coverage and limited master-worker 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 master and worker is not strict due to the underlying component architecture, placing workers near the domains they are expected to crawl can considerably reduce network load because of domain locality. In fact, the architecture 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 these two.

6.4 Perspectives for future work

While surpasses existing architectures such as [25, 33, 14] in terms of combined flexibility, ease of configurability and required development effort (see Table 6.1), the proposed architecture is not yet comprehensive. Below are 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 the 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 commercial databases provide are completely unnecessary in the 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 workers can offset the speed penalty due to unoptimized data access so that this is not a severe limitation in the system.

While the 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 master that has been introduced for the sake of coordination can undoubtedly increase the vulnerability of the system as a whole. Multiple coordinators that work in parallel to designate URLs to workers may improve the architecture in this respect.

Even though the proposed system features a graphical user interface, the current implementation is fairly rudimentary. It is capable of monitoring the current state of workers that attach to the user interface as well as fine-tuning their behavior by adjusting configuration settings, but the interface does not permit major on-the-fly restructuring of a running system. In particular, this means that the way components are interconnected can only be specified by means of XML descriptors. It would contribute to ease of use if this were possible directly via the user interface. Note that the architecture has full background support for this extension of the graphical user interface.

Finally, a large-scale crawl first with a simulation of the web based on its statistical characteristics available in research, second with a real crawl should be performed for comprehensiveness. Although smaller-scale crawls have been conducted, a large-scale traversal might reveal some deficiencies currently unseen in the system and will promote future development.

Bibliography

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

[2] *Asynchronous Programming Design Patterns*, .NET Framework Developer's Guide, *http://msdn2.microsoft.com/en-us/library/ms228969(VS.80).aspx*

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

[4] *Force-based algorithms*, *http://en.wikipedia.org/wiki/Force-based_algorithms*

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

[6] *The Open Directory Project*, *http://dmoz.org/*

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

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

[9] *SourceForge.net project site of Crawler.*NET, *http://sourceforge.net/projects/webcrawler*

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

[11] *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/*

[12] *The Yahoo! Directory*, *http://dir.yahoo.com/*

[13] 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, *http://gnunet.org/papers/p422-bloom.pdf*

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

[15] 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*

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

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

[18] 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*

[19] Junghoo Cho, Hector Garcia-Molina, *Parallel Crawlers*, WWW2002, Honolulu, Hawaii, May 7-11, 2002, ACM 1-58113-449-5/02/0005 *http://oak.cs.ucla.edu/ cho/papers/cho-parallel.pdf*

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

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

[22] Jeffrey E. F. Friedl, *Mastering Regular Expressions*, 2nd Edition, O'Reilly, July 2002, ISBN 059 6002 89 0

[23] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns*, Addison-Wesley Publishing Co., 1995, ISBN 020 1633 61 2

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

[25] Allan Heydon, Marc Najork, *Mercator: A Scalable, Extensible Web Crawler*, Compaq Systems Research Center, Palo Alto, CA *http://research.microsoft.com/~najork/mercator.pdf*

[26] HUNYADI Levente, PALLOS Péter, *Crawler.NET: A component-based distributed framework for web traversal*, Students' Scientific Conference (TDK), Budapest University of Technology and Economics, Faculty of Electrical Engineering and Information Technology, November 17, 2006, *http://w3.enternet.hu/hunyadi/Publications/CrawlerNET/Report.pdf*

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

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

[29] Marc NAJORK, Janet WIENER, *Breadth-first search crawling yields high-quality pages*, Compaq Systems Research Center, World Wide Web 10, May 2-5, 2001, Hong Kong, ACM 1-58113-348-0/01/0005, *http://www10.org/cdrom/papers/pdf/p208.pdf*

[30] M. O. RABIN, *Fingerprinting by Random Polynomials*, Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981

[31] Dave RAGGETT, Arnaud LE HORS, Ian JACOBS, *HTML 4.01 Specification*, W3C Recommendation, 24 December 1999, *http://www.w3.org/TR/html4/*

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

[33] 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, *http://cis.poly.edu/suel/papers/crawl.pdf*

[34] SZEREDI Péter, LUKÁCSY Gergely, BENKŐ Tamás, *Theory and Practice of the Semantic Web* (English edition to be published in 2007, translation based on: SZEREDI Péter et al., *A szemantikus világháló elmélete és gyakorlata*, TYPOTEX, Budapest, 2005, ISBN 963 9548 48 0)

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

[36] I. H. Witten, A. Moffat, T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Second Edition, Morgan Kaufmann, May, 1999, ISBN 155 8605 70 3

Appendix A

XML configuration interface

A major advantage of a loosely-coupled architecture is that constituents see one another through interfaces while the actual implementation remains invisible. Nevertheless, a system does not become operable unless implementor classes are instantiated behind class interfaces.

XML offers an excellent method of describing how components, connectors and providers are interconnected in a declarative way. An XML file is associated with each framework process. When the process is started, it parses the file and instantiates objects based on class names specified therein. Thereafter, it binds providers and connectors to respective other providers and components, verifying in each case whether the binding declared in the XML matches the interface expected by the implementation in the assembly, raising an exception on a mismatch. Thus, at compile-time, only an interface-level knowledge of bindings is required. It is not until run-time that implementations behind interfaces are bound.

Figure A.1 shows a simple initial configuration for a worker process in a master-worker scenario. This configuration initializes an *MSSQLClientDataProvider* and a *DnsResolver* provider instance (in the *Providers* section), five instances of the *Downloader* and a single instance of the *HtmlParser* component (in the *Components* section). The classes of the instances and the assemblies in which they reside are given in the *type* attribute, separated by a comma. Additionally, three connectors are brought into existence, two of which are remote connectors. Remote connectors consist of a local connector and a receiver or a sender component, respectively, but the latter are initialized behind the scenes and do not appear in the XML file. Notably, network addresses specified for remote connectors instruct receiver and sender components on 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 (in topological ordering), connectors and finally components.

```
<?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="AddressResolver"
         provider="reference:DnsResolver" />
      </Providers>
    </Component>
    <Component id="HyperlinkExtractor"
      type="Crawler.HtmlParser, CrawlerClient"
      source="reference:DownloadedDocuments"
      sink="reference:HyperlinksExtracted" />
  </Components>
</CrawlerSetup>
```

Figure A.1: A sample XML configuration file.

Appendix B

Implementing asynchronous components

Designing asynchronous components is seldom an easy task. As consuming input and producing output is not automatically synchronized, the component implementor has to ensure that data structures are not corrupted.

Figure B.1 shows an *AsynchronousComplexFilter* implementation. The component binds to two input connectors. The *InputUri* property associated with the first accepts URLs, 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 in accordance with the .NET asynchronous call model [2]. The second input connector is associated with the *InputIPAddress* property, which handles raw IP addresses that have to be passed on without any further processing. Note the presence of a lock mechanism. During the invocation of *AppendHostAddresses()*, the framework may call the *set* accessor of *InputIPAddress* or the *get* accessor of *Addresses*. In each case, 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, respectively, both manipulate the *address* field. The critical section guarantees that *addresses* is manipulated from a single thread at a time. The *SignalItemAvailable()* method notifies the framework that an item is available to be fed into the respective output connector.

```
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);
                }
                SignalItemAvailable("outputQueue");
            }
        }

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

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

Figure B.1: A sample *AsynchronousComplexFilter* implementation.

List of Abbreviations

Abbreviation	Expansion
ACID	Atomicity, Consistency, Isolation and Durability
BFS	Breadth-First Search
DFS	Depth-First Search
DNS	Domain Name System
FIFO	First In, First Out
GIOP	General Inter-ORB (Object Request Broker) Protocol
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
LIFO	Last In, First Out
NFS	Network File System
IP	Internet Protocol
PCRE	Perl-Compatible Regular Expressions
PDF	Portable Document Format
RTTI	RunTime Type Information
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	eXtensible Markup Language

List of Figures