CHAPTER 2

CRAWLING THE WEB

The World Wide Web, or the Web for short, is a collection of billions of documents written in a way that enables them to cite each other using *hyperlinks*, which is why they are a form of *hypertext*. These documents, or *Web pages*, are typically a few thousand characters long, written in a diversity of languages, and cover essentially all topics of human endeavor. Web pages are served through the Internet using the *hypertext transport protocol* (Http) to client computers, where they can be viewed using *browsers*. Http is built on top of the *transport control protocol* (TCP), which provides reliable data streams to be transmitted from one computer to another across the Internet.

Throughout this book, we shall study how automatic programs can analyze hypertext documents and the networks induced by the hyperlinks that connect them. To do so, it is usually necessary to fetch the pages to the computer where those programs will be run. This is the job of a *crawler* (also called a *spider*, *robot*, or *bot*). In this chapter we will study in detail how crawlers work. If you are more interested in how pages are indexed and analyzed, you can skip this chapter with hardly any loss of continuity.

I will assume that you have basic familiarity with computer networking using TCP, to the extent of writing code to open and close sockets and read and write data using a socket. We will focus on the organization of large-scale crawlers, which must handle millions of servers and billions of pages.

2.1 HTML and HTTP Basics

Web pages are written in a tagged markup language called the *hypertext markup language* (HTML). HTML lets the author specify layout and typeface, embed diagrams, and create hyperlinks. A hyperlink is expressed as an *anchor* tag with an `href` attribute, which names another page using a *uniform resource locator* (URL), like this:

```
<a href="http://www.cse.iitb.ac.in/">The IIT Bombay
Computer Science Department</a>
```

In its simplest form, the target URL contains a protocol field (http), a server hostname (*www.cse.iitb.ac.in*), and a file path (/, the "root" of the published file system).

A Web browser such as Netscape Communicator or Internet Explorer will let the reader *click* the computer mouse on the hyperlink. The click is translated transparently by the browser into a network request to fetch the target page using Http.

A browser will fetch and display a Web page given a complete URL like the one above, but to reveal the underlying network protocol, we will (ab)use the telnet command available on UNIX machines, as shown in Figure 2.1. First the telnet client (as well as any Web browser) has to resolve the server hostname *www.cse.iitb.ac.in* to an Internet address of the form 144.16.111.14 (called an *IP address*, IP standing for *Internet protocol*) to be able to contact the server using TCP. The mapping from name to address is done using the *Domain Name Service* (DNS), a distributed database of name-to-IP mappings maintained at known servers [202]. Next, the client connects to port 80, the default Http port, on the server. The underlined text is entered by the user (this is transparently provided by Web browsers). The *slanted* text is called the *MIME header*. (MIME stands for *multipurpose Internet mail extensions*, and is a metadata standard for email and Web content transfer.) The ends of the request and response headers are indicated by the sequence `CR-LF-CR-LF` (double newline, written in C/C++ code as `"\r\n\r\n"` and shown as the blank lines).

Browsing is a useful but restrictive means of finding information. Given a page with many links to follow, it would be unclear and painstaking to explore them in search of a specific information need. A better option is to *index* all the text so that information needs may be satisfied by keyword searches (as in library catalogs). To perform indexing, we need to fetch all the pages to be indexed using a crawler.


```
% telnet www.cse.iitb.ac.in 80
Trying 144.16.111.14...
Connected to www.cse.iitb.ac.in.
Escape character is '^]'.
GET / Http/1.0

Http/1.1 200 OK
Date: Sat, 13 Jan 2001 09:01:02 GMT
Server: Apache/1.3.0 (Unix) PHP/3.0.4
Last-Modified: Wed, 20 Dec 2000 13:18:38 GMT
ETag: "5c248-153d-3a40b1ae"
Accept-Ranges: bytes
Content-Length: 5437
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

<html>
<head><title>IIT Bombay CSE Department Home Page</title></head>
<body>...<a href="http://www.iitb.ac.in">IIT Bombay</a>...
</body></html>
Connection closed by foreign host.
%
```

FIGURE 2.1 Fetching a Web page using telnet and Http.

2.2 Crawling Basics

How does a crawler fetch "all" Web pages? Before the advent of the Web, traditional text collections such as bibliographic databases and journal abstracts were provided to the indexing system directly, say, on magnetic tape or disk. In contrast, there is no catalog of all accessible URLs on the Web. The only way to collect URLs is to scan collected pages for hyperlinks to other pages that have not been collected yet. This is the basic principle of crawlers. They start from a given set of URLs, progressively fetch and scan them for new URLs (*outlinks*), and then fetch these pages in turn, in an endless cycle. New URLs found thus represent potentially pending work for the crawler. The set of pending work expands quickly as the crawl proceeds, and implementers prefer to write this data to disk to relieve main memory as well as guard against data loss in the event of a crawler crash. There is no guarantee that all accessible Web pages will be located in

this fashion; indeed, the crawler may never halt, as pages will be added continually even as it is running. Apart from outlinks, pages contain text; this is submitted to a text indexing system (described in Section 3.1) to enable information retrieval using keyword searches.

It is quite simple to write a basic crawler, but a great deal of engineering goes into industry-strength crawlers that fetch a substantial fraction of all accessible Web documents. Web search companies like AltaVista, Northern Light, Inktomi, and the like do publish white papers on their crawling technologies, but piecing together the technical details is not easy. There are only a few documents in the public domain that give some detail, such as a paper about AltaVista's Mercator crawler [108] and a description of Google's first-generation crawler [26]. Based partly on such information, Figure 2.2 should be a reasonably accurate block diagram of a large-scale crawler.

The central function of a crawler is to fetch many pages at the same time, in order to overlap the delays involved in

1. Resolving the hostname in the URL to an IP address using DNS
2. Connecting a socket to the server and sending the request
3. Receiving the requested page in response

together with time spent in scanning pages for outlinks and saving pages to a local document repository. Typically, for short pages, DNS lookup and socket connection take a large portion of the processing time, which depends on round-trip times on the Internet and is generally unmitigated by buying more bandwidth.

The entire life cycle of a page fetch, as listed above, is managed by a logical *thread* of control. This need not be a thread or process provided by the operating system, but may be specifically programmed for this purpose for higher efficiency. In Figure 2.2 this is shown as the "Page fetching context/thread," which starts with DNS resolution and finishes when the entire page has been fetched via Http (or some error condition arises). After the fetch context has completed its task, the page is usually stored in compressed form to disk or tape and also scanned for outgoing hyperlinks (hereafter called "outlinks"). Outlinks are checked into a work pool. A load manager checks out enough work from the pool to maintain network utilization without overloading it. This process continues until the crawler has collected a "sufficient" number of pages. It is difficult to define "sufficient" in general. For an intranet of moderate size, a complete crawl may well be possible. For the Web, there are indirect estimates of the number

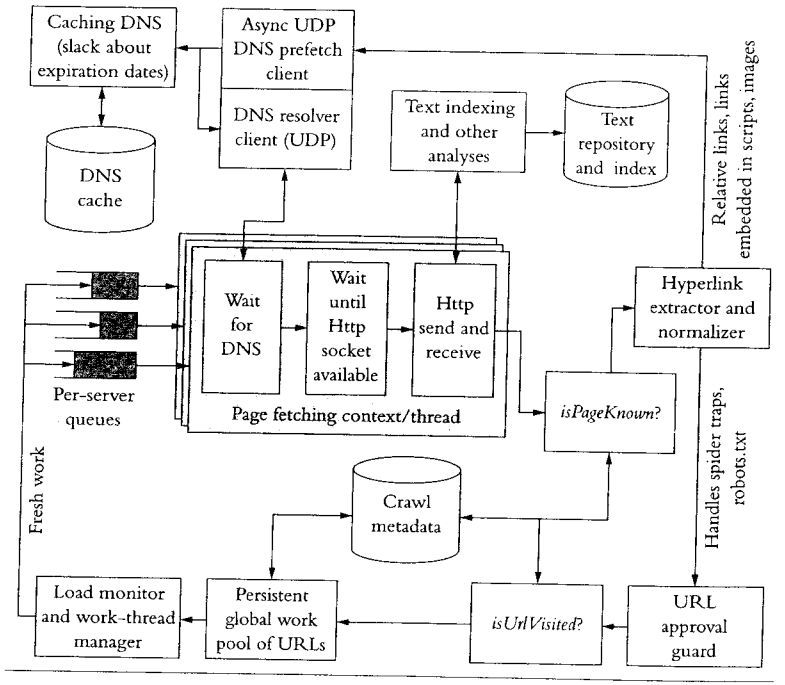

FIGURE 2.2 Typical anatomy of a large-scale crawler.

of publicly accessible pages, and a crawler may be run until a substantial fraction is fetched. Organizations with less networking or storage resources may need to stop the crawl for lack of space, or to build indices frequently enough to be useful.

2.3 Engineering Large-Scale Crawlers

In the previous section we discussed a basic crawler. Large-scale crawlers that send requests to millions of Web sites and collect hundreds of millions of pages need a great deal of care to achieve high performance. In this section we will discuss the important performance and reliability considerations for a large-scale crawler. Before we dive into the details, it will help to list the main concerns:

- Because a single page fetch may involve several seconds of network latency, it is essential to fetch many pages (typically hundreds to thousands) at the same time to utilize the network bandwidth available.
- Many simultaneous fetches are possible only if the DNS lookup is streamlined to be highly concurrent, possibly replicated on a few DNS servers.
- Multiprocessing or multithreading provided by the operating system is not the best way to manage the multiple fetches owing to high overheads. The best bet is to explicitly encode the state of a fetch context in a data structure and use asynchronous sockets, which do not block the process/thread using it, but can be polled to check for completion of network transfers.
- Care is needed to extract URLs and eliminate duplicates to reduce redundant fetches and to avoid "spider traps"—hyperlink graphs constructed carelessly or malevolently to keep a crawler trapped in that graph, fetching what can potentially be an infinite set of "fake" URLs.

2.3.1 DNS Caching, Prefetching, and Resolution

Address resolution is a significant bottleneck that needs to be overlapped with other activities of the crawler to maintain throughput. In an ordinary local area network, a DNS server running on a modest PC can perform name mappings for hundreds of workstations. A crawler is much more demanding as it may generate dozens of mapping requests per second. Moreover, many crawlers avoid fetching too many pages from one server, which might overload it; rather, they spread their access over many servers at a time. This lowers the locality of access to the DNS cache. For all these reasons, large-scale crawlers usually include a customized DNS component for better performance. This comprises a custom client for address resolution and possibly a caching server and a prefetching client.

First, the DNS caching server should have a large cache that should be persistent across DNS restarts, but residing largely in memory if possible. A desktop PC with 256 MB of RAM and a disk cache of a few GB will be adequate for a caching DNS, but it may help to have a few (say, two to three) of these. Normally, a DNS cache has to honor an expiration date set on mappings provided by its upstream DNS server or peer. For a crawler, strict adherence to expiration dates is not too important. (However, the DNS server should try to keep its mapping as up to date as possible by remapping the entries in cache during relatively idle time intervals.) Second, many clients for DNS resolution are coded poorly. Most UNIX systems provide an implementation of `gethostbyname` (the DNS client

API—application program interface), which cannot concurrently handle multiple outstanding requests. Therefore, the crawler cannot issue many resolution requests together and poll at a later time for completion of individual requests, which is critical for acceptable performance. Furthermore, if the system-provided client is used, there is no way to distribute load among a number of DNS servers. For all these reasons, many crawlers choose to include their own custom client for DNS name resolution. The Mercator crawler from Compaq System Research Center reduced the time spent in DNS from as high as 87% to a modest 25% by implementing a custom client. The ADNS asynchronous DNS client library[1] is ideal for use in crawlers.

In spite of these optimizations, a large-scale crawler will spend a substantial fraction of its network time not waiting for Http data transfer, but for address resolution. For every hostname that has not been resolved before (which happens frequently with crawlers), the local DNS may have to go across many network hops to fill its cache for the first time. To overlap this unavoidable delay with useful work, *prefetching* can be used. When a page that has just been fetched is parsed, a stream of `HREF`s is extracted. Right at this time, that is, even before any of the corresponding URLs are fetched, hostnames are extracted from the `HREF` targets, and DNS resolution requests are made to the caching server. The prefetching client is usually implemented using UDP (*user datagram protocol*, a connectionless, packet-based communication protocol that does not guarantee packet delivery) instead of TCP, and it does not wait for resolution to be completed. The request serves only to fill the DNS cache so that resolution will be fast when the page is actually needed later on.

2.3.2 Multiple Concurrent Fetches

Research-scale crawlers fetch up to hundreds of pages per second. Web-scale crawlers fetch hundreds to thousands of pages per second. Because a single download may take several seconds, crawlers need to open many socket connections to different Http servers at the same time. There are two approaches to managing multiple concurrent connections: using multithreading and using nonblocking sockets with event handlers. Since crawling performance is usually limited by network and disk, multi-CPU machines generally do not help much.

1. See *www.chiark.greenend.org.uk/~ian/adns/*.

Multithreading

After name resolution, each logical thread creates a client socket, connects the socket to the Http service on a server, sends the Http request header, then reads the socket (by calling `recv`) until no more characters are available, and finally closes the socket. The simplest programming paradigm is to use *blocking* system calls, which suspend the client process until the call completes and data is available in user-specified buffers.

This programming paradigm remains unchanged when each logical thread is assigned to a physical thread of control provided by the operating system, for example, through the `pthreads` multithreading library available on most UNIX systems [164]. When one thread is suspended waiting for a `connect`, `send`, or `recv` to complete, other threads can execute. Threads are not generated dynamically for each request; rather, a fixed number of threads is allocated in advance. These threads use a shared concurrent work-queue to find pages to fetch. Each thread manages its own control state and stack, but shares data areas. Therefore, some implementers prefer to use processes rather than threads so that a disastrous crash of one process does not corrupt the state of other processes.

There are two problems with the concurrent thread/process approach. First, mutual exclusion and concurrent access to data structures exact some performance penalty. Second, as threads/processes complete page fetches and start modifying the document repository and index concurrently, they may lead to a great deal of interleaved, random input-output on disk, which results in slow disk seeks.

The second performance problem may be severe. To choreograph disk access and to transfer URLs and page buffers between the work pool, threads, and the repository writer, the numerous fetching threads/processes must use one of shared memory buffers, interprocess communication, semaphores, locks, or short files. The exclusion and serialization overheads can become serious bottlenecks.

Nonblocking sockets and event handlers

Another approach is to use *nonblocking* sockets. With nonblocking sockets, a `connect`, `send`, or `recv` call returns immediately without waiting for the network operation to complete. The status of the network operation may be polled separately. In particular, a nonblocking socket provides the `select` system call, which lets the application suspend and wait until more data can be read from or written to the socket, timing out after a prespecified deadline. `select` can in fact monitor several sockets at the same time, suspending the calling process until *any one* of the sockets can be read or written.

Each active socket can be associated with a data structure that maintains the state of the logical thread waiting for some operation to complete on that socket, and callback routines that complete the processing once the fetch is completed. When a select call returns with a socket identifier, the corresponding state record is used to continue processing. The data structure also contains the page in memory as it is being fetched from the network. This is not very expensive in terms of RAM. One thousand concurrent fetches on 10 KB pages would still use only 10 MB of RAM.

Why is using `select` more efficient? The completion of page fetching threads is *serialized*, and the code that completes processing the page (scanning for outlinks, saving to disk) is not interrupted by other completions (which may happen but are not detected until we explicitly `select` again). Consider the pool of freshly discovered URLs. If we used threads or processes, we would need to protect this pool against simultaneous access with some sort of mutual exclusion device. With `select`s, there is no need for locks and semaphores on this pool. With processes or threads writing to a sequential dump of pages, we need to make sure disk `write`s are not interleaved. With `select`, we only append complete pages to the log, again without the fear of interruption.

2.3.3 Link Extraction and Normalization

It is straightforward to search an HTML page for hyperlinks, but URLs extracted from crawled pages must be processed and filtered in a number of ways before throwing them back into the work pool. It is important to clean up and canonicalize URLs so that pages known by different URLs are not fetched multiple times. However, such duplication cannot be eliminated altogether, because the mapping between hostnames and IP addresses is many-to-many, and a "site" is not necessarily the same as a "host."

A computer can have many IP addresses and many hostnames. The reply to a DNS request includes an IP address and a *canonical hostname*. For large sites, many IP addresses may be used for load balancing. Content on these hosts will be mirrors, or may even come from the same file system or database. On the other hand, for organizations with few IP addresses and a need to publish many logical sites, *virtual hosting* or *proxy pass* may be used[2] to map many different sites (hostnames) to a single IP address (but a browser will show different content for

2. See the documentation for the Apache Web server at *www.apache.org/*.

the different sites). The best bet is to avoid IP mapping for canonicalization and stick to the canonical hostname provided by the DNS response.

Extracted URLs may be *absolute* or *relative*. An example of an absolute URL is `http://www.iitb.ac.in/faculty/`, whereas a relative URL may look like `photo.jpg` or `/~soumen/`. Relative URLs need to be interpreted with reference to an absolute *base* URL. For example, the absolute form of the second and third URLs with regard to the first are `http://www.iitb.ac.in/faculty/photo.jpg` and `http://www.iitb.ac.in/~soumen/` (the starting "/" in `/~soumen/` takes you back to the root of the Http server's published file system). A completely canonical form including the default Http port (number 80) would be `http://www.iitb.ac.in:80/faculty/photo.jpg`.

Thus, a canonical URL is formed by the following steps:

1. A standard string is used for the protocol (most browsers tolerate Http, which should be converted to lowercase, for example).
2. The hostname is canonicalized as mentioned above.
3. An explicit port number is added if necessary.
4. The path is normalized and cleaned up, for example, `/books/../papers/sigmod1999.ps` simplifies to `/papers/sigmod1999.ps`.

2.3.4 Robot Exclusion

Another necessary step is to check whether the server prohibits crawling a normalized URL using the `robots.txt` mechanism. This file is usually found in the Http root directory of the server (such as `http://www.iitb.ac.in/robots.txt`). This file specifies a list of path prefixes that crawlers should *not* attempt to fetch. The `robots.txt` file is meant for crawlers only and does not apply to ordinary browsers. This distinction is made based on the `User-agent` specification that clients send to the Http server (but this can be easily spoofed). Figure 2.3 shows a sample `robots.txt` file.

2.3.5 Eliminating Already-Visited URLs

Before adding a new URL to the work pool, we must check if it has already been fetched at least once, by invoking the *isUrlVisited?* module, shown in Figure 2.2. (Refreshing the page contents is discussed in Section 2.3.11.) Many sites are quite densely and redundantly linked, and a page is reached via many paths; hence, the *isUrlVisited?* check needs to be very quick. This is usually achieved by computing a hash function on the URL.


```
# AltaVista Search
User-agent: AltaVista Intranet V2.0 W3C Webreq
Disallow: /Out-Of-Date

# exclude some access-controlled areas
User-agent: *
Disallow: /Team
Disallow: /Project
Disallow: /Systems
```

FIGURE 2.3 A sample `robots.txt` file.

For compactness and uniform size, canonical URLs are usually hashed using a hash function such as MD5. (The MD5 algorithm takes as input a message of arbitrary length and produces as output a 128-bit *fingerprint* or *message digest* of the input. It is conjectured that it is computationally hard to produce two messages having the same message digest, or to produce any message having a prespecified message digest value. See *www.rsasecurity.com/rsalabs/faq/3-6-6.html* for details.) Depending on the number of distinct URLs that must be supported, the MD5 may be collapsed into anything between 32 and 128 bits, and a database of these hash values is maintained. Assuming each URL costs just 8 bytes of hash value (ignoring search structure costs), a billion URLs will still cost 8 GB, a substantial amount of storage that usually cannot fit in main memory.

Storing the set of hash values on disk unfortunately makes the *isUrlVisited?* check slower, but luckily, there is some locality of access on URLs. Some URLs (such as *www.netscape.com/*) seem to be repeatedly encountered no matter which part of the Web the crawler is traversing. Thanks to relative URLs within sites, there is also some spatiotemporal locality of access: once the crawler starts exploring a site, URLs within the site are frequently checked for a while.

To exploit locality, we cannot hash the whole URL to a single hash value, because a good hash function will map the domain strings uniformly over the range. This will jeopardize the second kind of locality mentioned above, because paths on the same host will be hashed over the range uniformly. This calls for a two-block or two-level hash function. The most significant bits (say, 24 bits) are derived by hashing the hostname plus port only, whereas the lower-order bits (say, 40 bits) are derived by hashing the path. The hash values of URLs on the same host will therefore match in the 24 most significant bits. Therefore, if

the concatenated bits are used as a key in a B-tree that is cached at page level, spatiotemporal locality is exploited.

Finally, the qualifying URLs (i.e., those whose hash values are not found in the B-tree) are added to the pending work set on disk, also called the *frontier* of the crawl. The hash values are also added to the B-tree.

2.3.6 Spider Traps

Because there is no editorial control on Web content, careful attention to coding details is needed to render crawlers immune to inadvertent or malicious quirks in sites and pages. Classic lexical scanning and parsing tools are almost useless. I have encountered a page with 68 KB of null characters in the middle of a URL that crashed a lexical analyzer generated by flex.[3] Hardly any page follows the HTML standard to a level where a context-free parser like yacc or bison can parse it well. Commercial crawlers need to protect themselves from crashing on ill-formed HTML or misleading sites. HTML scanners have to be custom-built to handle errors in a robust manner, discarding the page summarily if necessary.

Using soft directory links and path remapping features in an Http server, it is possible to create an infinitely "deep" Web site, in the sense that there are paths of arbitrary depth (in terms of the number of slashes in the path or the number of characters). CGI (*common gateway interface*) scripts can be used to generate an infinite number of pages dynamically (e.g., by embedding the current time or a random number). A simple check for URL length (or the number of slashes in the URL) prevents many "infinite site" problems, but even at finite depth, Http servers can generate a large number of dummy pages dynamically. The following are real URLs encountered in a recent crawl:

- *www.troutbums.com/Flyfactory/hatchline/hatchline/hatchline/flyfactory/flyfactory/hatchline/flyfactory/flyfactory/flyfactory/flyfactory/flyfactory/flyfactory/flyfactory/flyfactory/hatchline/hatchline/flyfactory/flyfactory/hatchline/*
- *www.troutbums.com/Flyfactory/flyfactory/flyfactory/hatchline/hatchline/flyfactory/hatchline/flyfactory/hatchline/flyfactory/flyfactory/flyfactory/hatchline/flyfactory/hatchline/*
- *www.troutbums.com/Flyfactory/hatchline/hatchline/flyfactory/flyfactory/flyfactory/flyfactory/hatchline/flyfactory/flyfactory/flyfactory/flyfactory/flyfactory/flyfactory/flyfactory/hatchline/*

3. Available online at *www.gnu.org/software/flex/*.

Certain classes of traps can be detected (see the following section), but no automatic technique can be foolproof. The best policy is to prepare regular statistics about the crawl. If a site starts dominating the collection, it can be added to the *guard* module shown in Figure 2.2, which will remove from consideration any URL from that site. Guards may also be used to disable crawling active content such as CGI form queries, or to eliminate URLs whose data types are clearly not textual (e.g., not one of HTML, plain text, PostScript, PDF, or Microsoft Word).

2.3.7 Avoiding Repeated Expansion of Links on Duplicate Pages

It is desirable to avoid fetching a page multiple times under different names (e.g., u_1 and u_2), not only to reduce redundant storage and processing costs but also to avoid adding a relative outlink v multiple times to the work pool as u_1/v and u_2/v. Even if u_1 and u_2 have been fetched already, we should control the damage at least at this point. Otherwise there could be quite a bit of redundancy in the crawl, or worse, the crawler could succumb to the kind of spider traps illustrated in the previous section.

Duplicate detection is essential for Web crawlers owing to the practice of *mirroring* Web pages and sites—that is, copying them to a different host to speed up access to a remote user community. If u_1 and u_2 are exact duplicates, this can be detected easily. When the page contents are stored, a digest (e.g., MD5) is also stored in an index. When a page is crawled, its digest is checked against the index (shown as *isPageKnown?* in Figure 2.2). This can be implemented to cost one seek per test. Another way to catch such duplicates is to take the contents of pages u_1 and u_2, hash them to $h(u_1)$ and $h(u_2)$, and represent the relative link v as tuples $(h(u_1), v)$ and $(h(u_2), v)$. If u_1 and u_2 are aliases, the two outlink representations will be the same, and we can avoid the *isPageKnown?* implementation.

Detecting exact duplicates this way is not always enough, because mirrors may have minor syntactic differences, for example, the date of update, or the name and email of the site administrator may be embedded in the page. Unfortunately, even a single altered character will completely change the digest. *Shingling*, a more complex and robust way to detect *near* duplicates, is described in Section 3.3.2. Shingling is also useful for eliminating annoying duplicates from search engine responses.

2.3.8 Load Monitor and Manager

Network requests are orchestrated by the load monitor and thread manager shown in Figure 2.2. The load monitor keeps track of various system statistics:

- Recent performance of the *wide area network* (WAN) connection, say, latency and bandwidth estimates. Large crawlers may need WAN connections from multiple *Internet service providers* (ISPs); in such cases their performance parameters are individually monitored.
- An operator-provided or estimated maximum number of open sockets that the crawler should not exceed.
- The current number of active sockets.

The load manager uses these statistics to choose units of work from the pending work pool or frontier, schedule the issue of network resources, and distribute these requests over multiple ISPs if appropriate.

2.3.9 Per-Server Work-Queues

Many commercial Http servers safeguard against *denial of service* (DoS) attacks. DoS attackers swamp the target server with frequent requests that prevent it from serving requests from bona fide clients. A common first line of defense is to limit the speed or frequency of responses to any fixed client IP address (to, say, at most three pages per second). Servers that have to execute code in response to requests (e.g., search engines) are even more sensitive; frequent requests from one IP address are in fact actively penalized.

As an Http client, a crawler needs to avoid such situations, not only for high performance but also to avoid legal action. Well-written crawlers limit the number of active requests to a given server IP address at any time. This is done by maintaining a queue of requests for each server (see Figure 2.2). Requests are removed from the head of the queue, and network activity is initiated at a specified maximum rate. This technique also reduces the exposure to spider traps: no matter how large or deep a site is made to appear, the crawler fetches pages from it at some maximum rate and distributes its attention relatively evenly between a large number of sites.

From version 1.1 onward, Http has defined a mechanism for opening one connection to a server and keeping it open for several requests and responses in succession. Per-server host queues are usually equipped with Http version 1.1 persistent socket capability. This reduces overheads of DNS access and Http connection setup. On the other hand, to be polite to servers (and also because servers protect themselves by closing the connection after some maximum number of transfers), the crawler must move from server to server often. This tension between access locality and politeness (or protection against traps) is inherent in designing crawling policies.

2.3.10 Text Repository

The crawler's role usually ends with dumping the contents of the pages it fetches into a repository. The repository can then be used by a variety of systems and services which may, for instance, build a keyword index on the documents (see Chapter 3), classify the documents into a topic directory like Yahoo! (see Chapter 5), or construct a hyperlink graph to perform link-based ranking and social network analysis (see Chapter 7). Some of these functions can be initiated within the crawler itself without the need for preserving the page contents, but implementers often prefer to decouple the crawler from these other functions for efficiency and reliability, provided there is enough storage space for the pages. Sometimes page contents need to be stored to be able to provide, along with responses, short blurbs from the matched pages that contain the query terms.

Page-related information is stored in two parts: metadata and page contents. The metadata includes fields like content type, last modified date, content length, Http status code, and so on. The metadata is relational in nature but is usually managed by custom software rather than a relational database. Conventional relational databases pay some overheads to support concurrent updates, transactions, and recovery. These features are not needed for a text index, which is usually managed by bulk updates with permissible downtime.

HTML page contents are usually stored compressed using, for example, the popular compression library zlib. Since the typical text or HTML Web page is 10 KB long[4] and compresses down to 2 to 4 KB, using one file per crawled page is ruled out by file block fragmentation (most file systems have a 4 to 8 KB file block size). Consequently, page storage is usually relegated to a custom storage manager that provides simple access methods for the crawler to add pages and for programs that run subsequently (e.g., the indexer) to retrieve documents.

For small-scale systems where the repository is expected to fit within the disks of a single machine, one may use the popular public domain storage manager Berkeley DB (available from *www.sleepycat.com/*), which manages disk-based databases within a single file. Berkeley DB provides several access methods. If pages need to be accessed using a key such as their URLs, the database can

4. Graphic files may be longer.

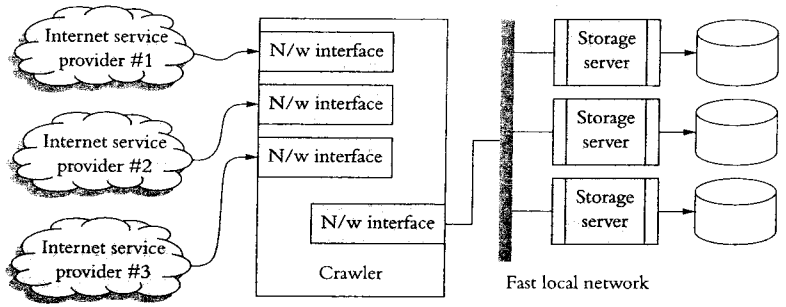

FIGURE 2.4 Large-scale crawlers often use multiple ISPs and a bank of local storage servers to store the pages crawled.

be configured as a hash table or a B-tree, but updates will involve expensive disk seeks, and a fragmentation loss between 15% and 25% will accrue. If subsequent page processors can handle pages in any order, which is the case with search engine indexing, the database can be configured as a sequential log of page records. The crawler only appends to this log, which involves no seek and negligible space overhead. It is also possible to first concatenate several pages and then compress them for a better compression factor.

For larger systems, the repository may be distributed over a number of storage servers connected to the crawler through a fast local network (such as gigabit Ethernet), as shown in Figure 2.4. The crawler may hash each URL to a specific storage server and send it the URL and the page contents. The storage server simply appends it to its own sequential repository, which may even be a tape drive, for archival. High-end tapes can transfer over 40 GB per hour,[5] which is about 10 million pages per hour, or about 200 hours for the whole Web (about 2 billion pages) at the time of writing. This is comparable to the time it takes today for the large Web search companies to crawl a substantial portion of the Web. Obviously, to complete the crawl in as much time requires the aggregate network bandwidth to the crawler to match the 40 GB per hour number, which is about 100 Mb per second, which amounts to about two T3-grade leased lines.

5. I use B for byte and b for bit.


```
% telnet www.cse.iitb.ac.in 80
Trying 144.16.111.14...
Connected to surya.cse.iitb.ac.in.
Escape character is '^]'.
GET / HTTP/1.0
If-modified-since: Sat, 13 Jan 2001 09:01:02 GMT

HTTP/1.1 304 Not Modified
Date: Sat, 13 Jan 2001 10:48:58 GMT
Server: Apache/1.3.0 (Unix) PHP/3.0.4
Connection: close
ETag: "5c248-153d-3a40b1ae"
Connection closed by foreign host.
%
```

FIGURE 2.5 Using the `If-modified-since` request header to check if a page needs to be crawled again. In this specific case it does not.

2.3.11 Refreshing Crawled Pages

Ideally, a search engine's index should be *fresh*—that is, it should reflect the most recent version of all documents crawled. Because there is no general mechanism of updates and notifications, the ideal cannot be attained in practice. In fact, a Web-scale crawler never "completes" its job; it is simply stopped when it has collected "enough" pages. Most large search engines then index the collected pages and start a fresh crawl. Depending on the bandwidth available, a round of crawling may run up to a few weeks. Many crawled pages do not change during a round—or ever, for that matter—but some sites may change many times.

Figure 2.5 shows how to use the Http protocol to check if a page changed since a specified time and, if so, to fetch the page contents. Otherwise the server sends a "not modified" response code and does not send the page. For a browser this may be useful, but for a crawler it is not as helpful, because, as I have noted, resolving the server address and connecting a TCP socket to the server already take a large chunk of crawling time.

When a new crawling round starts, it would clearly be ideal to know which pages have changed since the last crawl and refetch only those pages. This is possible in a very small number of cases, using the `Expires` Http response header (see Figure 2.6). For each page that did not come with an expiration date, we have to guess if revisiting that page will yield a modified version. If the crawler

```
% telnet vancouver-webpages.com 80
Trying 216.13.169.244...
Connected to vancouver-webpages.com (216.13.169.244).
Escape character is '^]'.
HEAD/cgi-pub/cache-test.pl/exp=in+1+minute&mod=Last+Night&rfc=1123 HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 26 Feb 2002 04:56:09 GMT
Server: Apache/1.3.6 (Unix)  (Red Hat/Linux) mod_perl/1.19
Expires: Tue, 26 Feb 2002 04:57:10 GMT
Last-Modified: Tue, 26 Feb 2002 04:56:10 GMT
Connection: close
Content-Type: text/html
```

FIGURE 2.6 Some sites with time-sensitive information send an `Expires` attribute in the Http response header.

had access to some sort of score reflecting the probability that each page has been modified, it could simply fetch URLs in decreasing order of that score. Even a crawler that runs continuously would benefit from an estimate of the expiration date of each page that has been crawled.

We can build such an estimate by assuming that the recent update rate will remain valid for the next crawling round—that is, that the recent past predicts the future. If the average interval at which the crawler checks for changes is smaller than the intermodification times of a page, we can build a reasonable estimate of the time to the next modification. The estimate could be way off, however, if the page is changed more frequently than the poll rate: we might have no idea how many versions successive crawls have missed. Another issue is that in an expanding Web, more pages appear young as time proceeds. These issues are discussed by Brewington and Cybenko [24], who also provide algorithms for maintaining a crawl in which most pages are fresher than a specified epoch. Cho [50] has also designed incremental crawlers based on the same basic principle.

Most search engines cannot afford to wait for a full new round of crawling to update their indices. Between every two complete crawling rounds, they run a crawler at a smaller scale to monitor fast-changing sites, especially related to current news, weather, and the like, so that results from this index can be patched into the master index. This process is discussed in Section 3.1.2.

2.4 Putting Together a Crawler

The World Wide Web Consortium (*www.w3c.org/*) has published a reference implementation of the Http client protocol in a package called `w3c-libwww`. It is written in C and runs on most operating systems. The flexibility and consequent complexity of the API may be daunting, but the package greatly facilitates the writing of reasonably high-performance crawlers. Commercial crawlers probably resemble crawlers written using this package up to the point where storage management begins.

Because the details of commercial crawlers are carefully guarded, I will focus on the design and use of the `w3c-libwww` library instead. This section has two parts. In the first part, I will discuss the internals of a crawler built along the same style as `w3c-libwww`. Since `w3c-libwww` is large, general, powerful, and complex, I will abstract its basic structure through pseudocode that uses C++ idioms for concreteness. In the second part, I will give code fragments that show how to use `w3c-libwww`.

2.4.1 Design of the Core Components

It is easiest to start building a crawler with a core whose only responsibility is to copy bytes from network sockets to storage media: this is the `Crawler` class. The `Crawler`'s contract with the user is expressed in these methods:

```
class Crawler {
  void setDnsTimeout(int milliSeconds);
  void setHttpTimeout(int milliSeconds);
  void fetchPush(const string& url);
  virtual boolean fetchPull(string& url); // set url, return success
  virtual void fetchDone(const string& url,
    const ReturnCode returnCode, // timeout, server not found, ...
    const int httpStatus,        // 200, 404, 302, ...
    const hash_map<string, string>& mimeHeader,
      // Content-type = text/html
      // Last-modified = ...
    const unsigned char * pageBody,
    const size_t pageSize);
};
```

The user can `push` a URL to be fetched to the `Crawler`. The crawler implementation will guarantee that within a finite time (preset by the user using

setDnsTimeout and setHttpTimeout) the termination callback handler fetchDone will be called with the same URL and associated fields as shown. (I am hiding many more useful arguments for simplicity.) fetchPush inserts the URL into a memory buffer: this may waste too much memory for a Web-scale crawler and is volatile. A better option is to check new URLs into a persistent database and override fetchPull to extract new work from this database. The user also overrides the (empty) fetchDone method to process the document, usually storing page data and metadata from the method arguments, scanning pageBody for outlinks, and recording these for later fetchPulls. Other functions are implemented by extending the Crawler class. These include retries, redirections, and scanning for outlinks. In a way, "Crawler" is a misnomer for the core class; it just fetches a given list of URLs concurrently.

Let us now turn to the implementation of the Crawler class. We will need two helper classes called DNS and Fetch. Crawler is started with a fixed set of DNS servers. For each server, a DNS object is created. Each DNS object creates a UDP socket with its assigned DNS server as the destination. The most important data structure included in a DNS object is a list of Fetch contexts waiting for the corresponding DNS server to respond:

```
class DNS {
  list<Fetch*> waitForDns;
  :
  : //other members
}
```

A Fetch object contains the context required to complete the fetch of one URL using asynchronous sockets. waitForDns is the list of Fetches waiting for this particular DNS server to respond to their address-resolution requests.

Apart from members to hold request and response data and methods to deal with socket events, the main member in a Fetch object is a state variable that records the current stage in retrieving a page:

```
typedef enum {
  STATE_ERROR = -1, STATE_INIT = 0,
  STATE_DNS_RECEIVE, STATE_HTTP_SEND, STATE_HTTP_RECEIVE, STATE_FINAL
} State;
State state;
```


For completeness I also list a set of useful ReturnCodes. Most of these are self-explanatory; others have to do with the innards of the DNS and Http protocols.

```
typedef enum {
  SUCCESS = 0,
  //-----------------------------------------------------------------------
  DNS_SERVER_UNKNOWN,
  DNS_SOCKET, DNS_CONNECT, DNS_SEND, DNS_RECEIVE, DNS_CLOSE, DNS_TIMEOUT,
  // and a variety of error codes DNS_PARSE_... if the DNS response
  // cannot be parsed properly for some reason
  //-----------------------------------------------------------------------
  HTTP_BAD_URL_SYNTAX, HTTP_SERVER_UNKNOWN,
  HTTP_SOCKET, HTTP_CONNECT, HTTP_SEND, HTTP_RECEIVE,
  HTTP_TIMEOUT, HTTP_PAGE_TRUNCATED,
  //-----------------------------------------------------------------------
  MIME_MISSING, MIME_PAGE_EMPTY, MIME_NO_STATUS_LINE,
  MIME_UNSUPPORTED_HTTP_VERSION, MIME_BAD_CHUNK_ENCODING
} ReturnCode;
```

The remaining important data structures within the Crawler are given below.

```
class Crawler {
  deque<string> waitForIssue;
    // Requests wait here to limit the number of network connections.
    // When resources are available, they go to...
  hash_map<SocketID, DNS*> dnsSockets;
    // There is one entry for each DNS socket, i.e., for each DNS server.
    // New Fetch record entries join the shortest list.
    // Once addresses are resolved, Fetch records go to...
  deque<Fetch*> waitForHttp;
    // When the system can take on a new Http connection, Fetch records
    // move from waitForHttp to...
  hash_map<SocketID, Fetch*> httpSockets;
    // A Fetch record completes its lifetime while attached to an Http socket.
    // To avoid overloading a server, we keep a set of IP addresses that
    // we are nominally connected to at any given time
  hash_set<IPAddr> busyServers;
  :
  : //rest of Crawler definition
}
```

```
1:  Crawler::start()
2:  while event loop has not been stopped do
3:    if not enough active Fetches to keep system busy then
4:      try a fetchPull to replenish the system with more work
5:      if no pending Fetches in the system then
6:        stop the event loop
7:      end if
8:    end if
9:    if not enough Http sockets busy and
      there is a Fetch f in waitForHttp whose server IP address ∉ busyServers then
10:     remove f from waitForHttp
11:     lock the IP address by adding an entry to busyServers (to be polite
        to the server)
12:     change f.state to STATE_HTTP_SEND
13:     allocate an Http socket s to start the Http protocol
14:     set the Http timeout for f
15:     set httpSockets[s] to the Fetch pointer
16:     continue the outermost event loop
17:   end if
18:   if the shortest waitForDns is "too short" then
19:     remove a URL from the head of waitForIssue
20:     create a Fetch object f with this URL
21:     issue the DNS request for f
22:     set f.state to STATE_DNS_RECEIVE
23:     set the DNS timeout for f
24:     put f on the laziest DNS's waitForDns
25:     continue the outermost event loop
26:   end if
27:   collect open SocketIDs from dnsSockets and httpSockets
28:   also collect the earliest deadline over all active Fetches
29:   perform the select call on the open sockets with the earliest deadline
      as timeout
```

FIGURE 2.7 The Crawler's event loop. For simplicity, the normal workflow is shown, hiding many conditions where the state of a Fetch ends in an error.

The heart of the Crawler is a method called Crawler::start(), which starts its *event loop*. This is the most complex part of the Crawler and is given as a pseudocode in Figure 2.7. Each Fetch object passes through a workflow. It first


```
30:   if select returned with a timeout then
31:     for each expired Fetch record f in dnsSockets and httpSockets do
32:       remove f
33:       invoke f.fetchDone(...) with suitable timeout error codes
34:       remove any locks held in busyServers
35:     end for
36:   else
37:     find a SocketID s that is ready for read or write
38:     locate a Fetch record f in dnsSockets or httpSockets that was waiting on s
39:     if a DNS request has been completed for f then
40:       move f from waitForDns to waitForHttp
41:     else
42:       if socket is ready for writing then
43:         send request
44:         change f.state to STATE_HTTP_RECEIVE
45:       else
46:         receive more bytes
47:         if receive completed then
48:           invoke f.fetchDone(...) with successful status codes
49:           remove any locks held in busyServers
50:           remove f from waitForHttp and destroy f
51:         end if
52:       end if
53:     end if
54:   end if
55: end while
```

FIGURE 2.7 *(continued)*

joins the waitForDNS queue on some DNS object. When the server name resolution step is completed, it goes into the waitForHttp buffer. When we can afford another Http connection, it leaves waitForHttp and joins the httpSockets pool, where there are two major steps: sending the Http request and filling up a byte buffer with the response. Finally, when the page content is completely received, the user callback function fetchDone is called with suitable status information. The user has to extend the Crawler class and redefine fetchDone to parse the page and extract outlinks to make it a real crawler.

2.4.2 Case Study: Using w3c-libwww

So far we have seen a simplified account of how the internals of a package like w3c-libwww is designed; now we will see how to use it. The w3c-libwww API is extremely flexible and therefore somewhat complex, because it is designed not only for writing crawlers but for general, powerful manipulation of distributed hypertext, including text-mode browsing, composing dynamic content, and so on. Here we will sketch a simple application that issues a batch of URLs to fetch and installs a fetchDone callback routine that just throws away the page contents. We start with the main routine in Figure 2.8.

Unlike the simplified design presented in the previous section, w3c-libwww can process responses as they are streaming in and does not need to hold them in a memory buffer. The user can install various processors through which the incoming stream has to pass. For example, we can define a handler called hrefHandler to extract HREFs, which would be useful in a real crawler. It is registered with the w3c-libwww system as shown in Figure 2.8. Many other objects are mentioned in the code fragment below, but most of them are not key to understanding the main idea. hrefHandler is shown in Figure 2.9.

The method fetchDone, shown in Figure 2.10, is quite trivial in our case. It checks if the number of outstanding fetches is enough to keep the system busy; if not, it adds some more work. Then it just frees up resources associated with the request that has just completed and returns. Each page fetch is associated with an HTRequest object, similar to our Fetch object. At the very least, a termination handler should free this request object. If there is no more work to be found, it stops the event loop.

2.5 Bibliographic Notes

Details of the TCP/IP protocol and its implementation can be found in the classic work by Stevens [202]. Precise specifications of hypertext-related and older network protocols are archived at the World Wide Web Consortium (W3C Web site *www.w3c.org/*). Web crawling and indexing companies are rather protective about the engineering details of their software assets. Much of the discussion of the typical anatomy of large crawlers is guided by an early paper discussing the crawling system [26] for Google, as well as a paper about the design of Mercator, a crawler written in Java at Compaq Research Center [108]. There are many public-domain crawling and indexing packages, but most of them do not handle


```
vector<string> todo;
int tdx = 0;
  //...global variables storing all the URLs to be fetched...
int inProgress=0;
  //...keep track of active requests to exit the event loop properly...
int main(int ac, char ** av) {
  HTProfile_newRobot("CSE@IITBombay", "1.0");
  HTNet_setMaxSocket(64); // ...keep at most 64 sockets open at a time
  HTHost_setEventTimeout(40000);  //...Http timeout is 40 seconds
  //...install the hrefHandler...
  HText_registerLinkCallback(hrefHandler);
  //...install our fetch termination handler...
  HTNet_addAfter(fetchDone, NULL, NULL, HT_ALL, HT_FILTER_LAST);
  //...read URL list from file...
  ifstream ufp("urlset.txt");
  string url;
  while ( ufp.good() && ( ufp >> url ) && url.size() > 0 )
    todo.push_back(url);
  ufp.close();
  //...start off the first fetch...
  if ( todo.empty() ) return;
  ++inProgress;
  HTRequest * request = HTRequest_new();
  HTAnchor * anchor = HTAnchor_findAddress(todo[tdx++].c_str());
  if ( YES == HTLoadAnchor(anchor, request) ) {
    //...and enter the event loop...
    HTEventList_newLoop();
  }
  //...control returns here when event loop is stopped
}
```

FIGURE 2.8 Sketch of the main routine for a crawler using w3c-libwww.

the scale that commercial crawlers do. w3c-libwww is an open implementation suited for applications of moderate scale.

Estimating the size of the Web has fascinated Web users and researchers alike. Because the Web includes dynamic pages and spider traps, it is not easy to even define its size. Some well-known estimates were made by Bharat and Bröder [16] and Lawrence and Lee Giles [133]. The Web continues to grow, but not

```
void hrefHandler(HText * text,
                 int element_number,
                 int attribute_number,
                 HTChildAnchor * anchor,
                 const BOOL * present,
                 const char ** value)
{
  if ( !anchor ) return;
  HTAnchor * childAnchor = HTAnchor_followMainLink((HTAnchor*)anchor);
  if ( !childAnchor ) return;
  char * childUrl = HTAnchor_address((HTAnchor*) childAnchor);
  //...add childUrl to work pool, or issue a fetch right now...
  HT_FREE(childUrl);
}
```

FIGURE 2.9 A handler that is triggered by `w3c-libwww` whenever an `HREF` token is detected in the incoming stream.

```
#define LIBWWW_BATCH_SIZE 16
        //...number of concurrent fetches...
int fetchDone(HTRequest * request, HTResponse * response,
              void * param, int status)
{
  if ( request == NULL ) return -1;
  //...replenish concurrent fetch pool if needed...
  while ( inProgress < LIBWWW_BATCH_SIZE && tdx < todo.size() ) {
    ++inProgress;
    string newUrl(todo[tdx]);
    ++tdx;
    HTRequest * nrq = HTRequest_new();
    HTAnchor * nax = HTAnchor_findAddress(newUrl.c_str());
    (void) HTLoadAnchor(nax, nrq);
  }
  //...process the just-completed fetch here...
  inProgress--;
  const bool noMoreWork = ( inProgress <= 0 );
  HTRequest_delete(request);
  if ( noMoreWork )
    HTEventList_stopLoop();
  return 0;
}
```

FIGURE 2.10 Page fetch completion handler for the `w3c-libwww`–based crawler.

at as heady a pace as in the late 1990s. Nevertheless, some of the most accessed sites change frequently. The Internet Archive (*www.archive.org/*) started to archive large portions of the Web in October 1996, in a bid to prevent most of it from disappearing into the past [120]. At the time of this writing, the archive has about 11 billion pages, taking up over 100 terabytes. Storage at such a scale is not unprecedented: a music radio station holds about 10,000 records, or about 5 terabytes of uncompressed data, and the U.S. Library of Congress contains about 20 million volumes, or an estimated 20 terabytes of text. The Internet Archive is available to researchers, historians, and scholars. An interface called the Wayback Machine lets users access old versions of archived Web pages.