University of Alberta

An Efficient Scheme to Remove Crawler Traffic from the Internet

by

Xiaoqin Yuan ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2002

0-612-69783-5

University of Alberta

Library Release Form

Name of Author: Xiaoqin Yuan

Title of Thesis: An Efficient Scheme to Remove Crawler Traffic from the Internet

Degree: Master of Science

Year this Degree Granted: 2002

Xiaoqin Yuan

Xiaoqin Yuan
Department of Computing Science
232 Athabasca Hall
University of Alberta
Edmonton, AB
Canada, T6G 2E8

Date: Mar. 08, 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **An Efficient Scheme to Remove Crawler Traffic from the Internet** submitted by Xiaoqin Yuan in partial fulfillment of the requirements for the degree of **Master of Science**.

M. H. MacGregor

Dr. Mike H. MacGregor
Supervisor

J. Harms

Dr. Janelle Harms
Co-Supervisor

Gerald Häubl

Dr. Gerald Haubl
External Examiner

Date: March 8 2002

Abstract

One of the first things that any Internet 'neophyte' learns is how to search for information using search engines. Search engines tackle the daunting task of categorizing myriads of documents on the Web by using web crawlers - 'search agents'. This method, however, has been shown to place a significant load on the Web servers as well as tax the underlying network infrastructure. We address the aforementioned problem by introducing an efficient indexing system based on active networks. Our approach employs strategically placed active routers that constantly monitor passing Internet traffic, analyze it, and then transmit the index data to a dedicated back-end repository. Therefore, our proposal obviates the need for Web crawlers and effectively eliminates their adverse effect on Web servers and network resources. Our simulations have shown that our active indexing system is up to 30% more efficient than current web crawler based techniques. It is also shown that, given a limited network bandwidth, our system achieves a better throughput introduced by human clients and clients get responses more quickly since more bandwidth is made available to human requests.

Acknowledgements

First and foremost. I must thank my two supervisors: Dr. Mike H. MacGregor and Dr. Janelle Harms. During the whole course of the project. Mike is always there to provide suggestions when I'm at a loss. and sometimes a kind reminder when I need a motivation. Thanks. Mike. There is no way I can do enough for what you have done to me. Similarly. the discussions with Janelle about the simulation benefit me a lot. I truly appreciate her going over my thesis several times and the invaluable suggestions. I would also like to thank Dr. Gerald Haubl for being the external examiner.

Special thanks to my husband for the encouragement. the support and the company of so many nights on the other side of the cyberspace.

Finally. I would like to thank my parents and sister for their support and their long waiting for my return home.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Description

The Web has been growing at a remarkable pace since the introduction of the first Web browser and server in 1991[15]. By the end of 1993 the Web accounted for 1% of the traffic on the Internet. However, with the explosion of the number of Web users and Web sites, the Web was already responsible for around 75% of Internet traffic in terms of bytes by 1998[5]. Conservative estimates suggest that the number of Uniform Resource Locators (URLs) available to the public has grown beyond a billion. This fact means that trying to find some information among a billion pages is tantamount to looking for a needle in a haystack. Search engines make it possible, given a set of strings, to find individual pages on the Web while *Portal* sites can be used to find general information about a topic at a coarser level of granularity.

Search engines gather pages from all over the world with the aid of crawlers. Crawlers behind a search engine go out to retrieve Web pages periodically no matter whether pages have been modified or not. Generally speaking, it takes the crawlers of a search engine anywhere from 2 weeks to one month to update its database of Web pages. However, studies show that a large percentage of pages remain unchanged for more than 4 months, which implies that an unchanged page could be fetched a few times by crawlers of the same search engine. Currently there are more than one thousand search engines in the world. Therefore, it is possible for a page to be retrieved a few thousand times by crawlers within a short time period. Eventually, it is the underlying network that takes the responsibility to deliver the retrieved Web pages from a Web server to a crawler. Crawlers have the potential to swamp the network by generating excess traffic on the Internet. This work proposes a novel network architecture based on active networks. aimed to eliminate the traffic caused by crawlers from the Internet and to build the indexing data that search engines require at the same time.

1.2 Overview of the Active Networks

The ability to quickly create. deploy and manage new services in response to user demands is the primary motivation behind active networking[6]. With current networks. the introduction of new services is a challenging task and involves considerable service-specific computation and switching. Active networking is a new framework in which the network is designed not only to forward packets. but also to be dynamically programmed in order to support per-user services[7]. In active networks. users can supply both data and code to the network. and the network transports data between end systems and may execute user code. whereas in traditional networks. user programs execute only at the end systems and the network transports user data between end systems.

An active network is composed of a set of connected nodes which may be not all active nodes[8]. The functionality of an active node is divided into three major components: the Node Operating System (NodeOS). the Execution Environment (EEs). and the Active Applications(AAs). The architecture of an active node is illustrated in Figure 1.1. Each active node runs a NodeOS and one or more EEs. The NodeOS is responsible for the allocation and scheduling of the node's resources such as link bandwidth. CPU cycles and storage. It hides from EEs the details of resource management and the effects of the existence of other EEs. An EE exports a programming interface or implements a virtual machine that interprets active packets it receives. Thus. an EE acts like a shell program in a general-purpose computing system. An AA is a program which implements an end-to-end service using a combination of

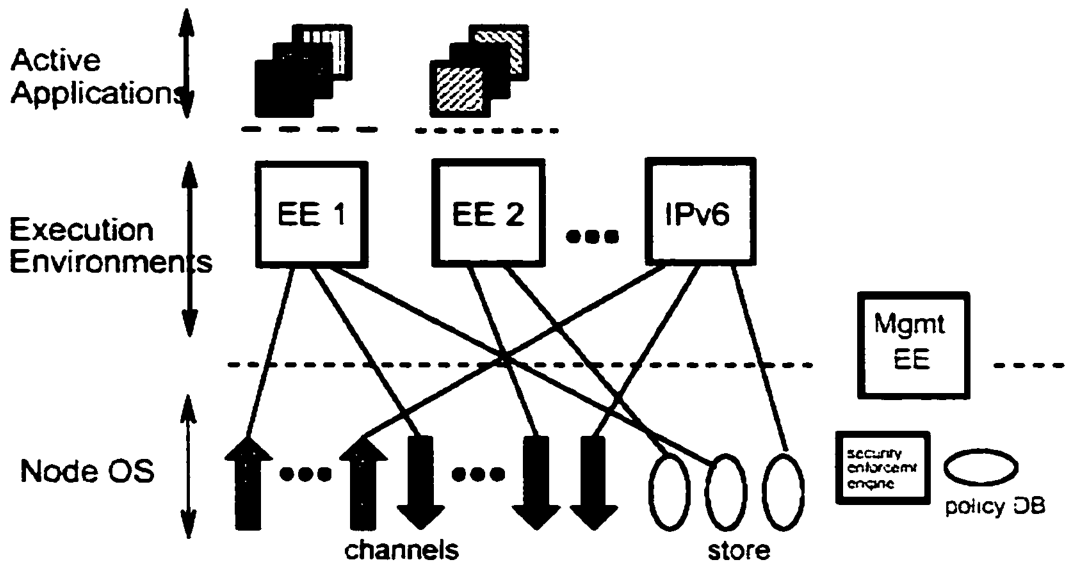


Figure 1.1: Active node architecture

packet forwarding and computation. It is the AA that implements customized services for end-users.

1.3 Contribution

1.3.1 Overview

To relieve Web servers and the Internet of the excessive load placed by Web crawlers. we introduce an efficient indexing system based on active networks. Our approach employs strategically placed active routers that constantly monitor passing Internet traffic. analyze it. and then transmit the resultant index data to a dedicated back-end repository. Therefore. our proposal obviates the need for Web crawlers and effectively eliminates their adverse effect on Web servers and the underlying network infrastructure. In order to see how much improvement is made by using our indexing scheme. We simulate two types of networks: the traditional network with crawler traffic and the active network without crawler traffic. but with the index data the active routers generate. We compare the network performance of the two systems.

1.3.2 Significance

First of all, by removing the need for Web crawlers, our scheme effectively removes the crawler traffic from the Internet and at the same time, builds the indexing information that search engines need in response to user queries. Given a limited network bandwidth, more bandwidth is made available to a variety of human clients and less congestion will happen. Therefore, the user-perceived latency is reduced since the Web traffic is moved faster between human visitors and Web servers. CPU cycles on the server side are also saved so that the server is dedicated to processing human clients' requests more efficiently.

The Web resources gathered by our system tend to be more complete than by crawlers. Currently, whether a Web page gets a chance to be fetched by crawlers relies on the link structure of the Web. Due to the strategic locations of gateway routers, all the Web pages get a chance to be captured by the system regardless of its link relation to other pages. This issue is further discussed in Section 3.3.1.

In addition to the *backlink count* and the *inverse document frequency* which are common statistical metrics used in ranking a user query's results, our system also can gather *document visiting frequency*, the number of times per unit time a page has been seen by its gateway. It gives us a dynamic behavior of a Web page. This statistic, if used in combination with static statistics such as *backlink count*, should make a user's desired results ranked higher.

Chapter 2

Traditional Web with Crawlers

Web clients and Web servers are two major software components of the Web. This chapter begins with a discussion of three different Web clients: a browser as the most popular form of a Web client. a spider which plays a key role in Web searching. and the less well known agent software[15]. Then we describe the operations of Web servers: the steps taken by a server to handle a request and three kinds of server architectures in use today: event-driven. process-driven. and hybrid server architecture. A server and a client communicate with each other using the Hypertext Transfer Protocol (HTTP). We will discuss two kinds of messages that HTTP carries: HTTP requests and HTTP responses. The current version of the HTTP protocol is HTTP/1.1 evolving from HTTP/1.0. There are many syntactic differences between HTTP/1.0 and HTTP/1.1 in terms of methods. headers and response codes. but we will only address the pipelining on persistent connections. the improvement in connection management introduced in HTTP/1.1. With the rapid increase of traffic on the Web. Web caching was the first technique designed to reduce user-perceived latency and reduce transmission of redundant traffic on the Internet. We describe the relevant header fields to maintain cache coherency in Section 5. As stated earlier. crawlers play a key part in searching. Section 6 discusses how crawlers are applied in indexing.

2.1 Web Clients

A Web client is a piece of software. It constructs a properly formatted Web request. establishes a connection and communicates with a Web server over a reliable transport-level connection.

2.1.1 Browser

Among the three forms of a client. the browser is the closest form to the user of the Internet. As a Web client. it constructs and sends an HTTP request. After receiving the response. it parses and tailors its display according to the user's configurations. Figure 2.1 demonstrates the various steps taken by a typical browser involved in a Web request[15]. In our example. we assume a user enters the Uniform Resource Locator (URL) *http://www.cnn.com/index.html* to the browser.

- The browser extracts the domain name *www.cnn.com* from the URL.
- The browser contacts one of the local DNS servers to convert *www.cnn.com* into an IP address. DNS(Domain Name System) translates hostnames into IP addresses and IP addresses into hostnames.
- After getting the destination IP address from the DNS server. the browser makes a TCP connection to port 80 on the destination Web server.
- The browser sends an HTTP request with the URL to the destination server to obtain the response.
- The browser parses the response and may set up additional connections to fetch embedded images if there are any. The additional connections may be established in parallel.

Not all the above steps are necessary for all requests due to caching and the implementation of persistent connections in HTTP/1.1. A cache is a local store of messages used to reduce the user's perceived latency in obtaining a response from a server. In the browser. we have two common kinds of caching.

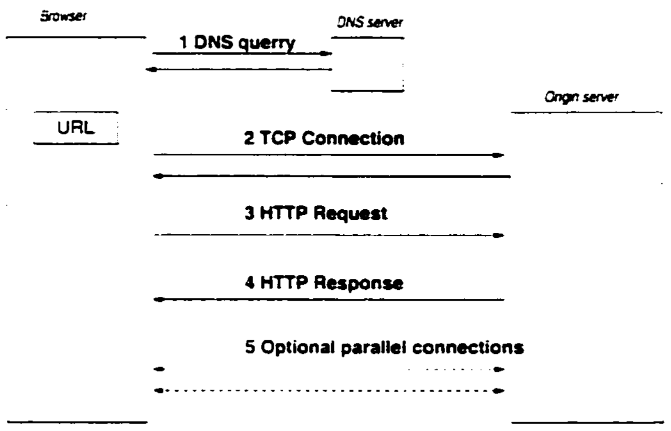


Figure 2.1: Steps in a browser session

A cache can be either stored in a part of memory of the running process or a part of the file system's disk space dedicated to caching. In HTTP/1.0. a separate TCP connection is established to fetch a single URL. If a Web page contains inline images. multiple connections are required to send requests to the server. which imposes extra load on both the server and the Internet. As opposed to HTTP/1.0. in HTTP/1.1. unless indicated in a HEADER field of the response from the server. the client assumes the server will maintain a persistent connection. Thus substantial resources are saved on both routers and hosts after the additional setups of the TCP connections are eliminated. In Section 4 of this chapter. we will address pipelining on persistent connections. an important feature in HTTP/1.1.

2.1.2 Crawlers

Crawlers are programs that follow links on Web sites to gather resources for search engines. We may see different names in the literature for crawlers such as *spiders. robots. wanderers. gatherers.* Search engines have become one of the most popular Web applications recently. The application of crawlers in search engines is the primary motivation for the creation of the Web crawlers. The resources the crawler gathers are used to generate indexing data. How crawlers are applied in the indexing will be explored in further depth in Section 6 of this chapter. Like any other Web client, crawlers construct an HTTP request to access resources at a site and parse the responses. The primary differences between a crawler and a browser are the dramatically higher number of sites contacted and requests sent and the absence of any display of the responses. Also, in practice, depending on the application the crawler is used for, only a portion of the resources might be fetched. Many crawlers ignore image or multimedia resources if the resources obtained are used to construct a searchable index of textual resources only.

How Crawlers Follow Links to Find Pages

- **Breadth-First Crawling** Breadth-first crawling obtains the starting page of a site and examines all the embedded hypertext references in it. For each of the references, it could fetch the corresponding page. All the links on the starting page will be retrieved before going further away. This is the most common way the crawlers follow links. Concurrent threads can implemented so that one thread is used to access one Web site and other threads go to other Web sites. Thus several pages are processed in parallel and the load is distributed among several hosts. Figure 2.2 shows the order of links to be followed when a breadth-first crawler indexes a site. In the graph, a small circle indicates a link. So if there are two circles in a rectangle, there are two links in the corresponding page.

- **Depth-First Crawling** An alternative approach, depth-first crawling, follows all the links from the first link on the starting page, then the first link on the second page, and so on. Once it has indexed the first link on each page, it indexes all the links on the last page. Then it *backbtracks* to the second last page and indexes the second link on that page. *Backtracking* happens when there is no deeper link to follow. It follows all the links from the second link on the second last page. It continues recursively until all the links following the first link on the starting page have been indexed. It then goes on to the second link and

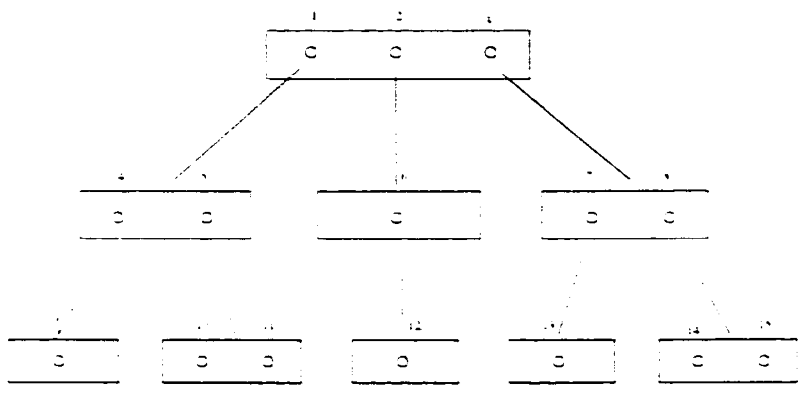


Figure 2.2: Breadth first search sequence

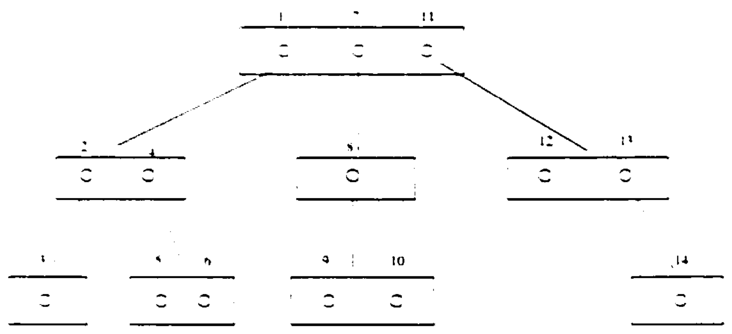


Figure 2.3: Depth first search sequence

subsequent links of the first page. Figure 2.3 shows the sequence of the links to be traversed when a depth-first crawler indexes a site.

Both breadth-first searching and depth-first-searching limit their depth of searching to a fixed number to save space and to ensure most valuable resources have been indexed. During the course of following links. if a crawler runs into a link which has been indexed. it will not index it again in order to avoid a loop.

Communicating with Crawlers

Some Web sites do not want to be bothered by crawlers. although some sites may benefit from indexing. However. it's difficult for a server to distinguish between clients since every request from a client is viewed as an independent request. There are typically two conventions that are followed by sites to have some control over crawlers visiting them[12].

- **Robots.txt** The first convention is at a site level. The Web site administrator maintains a plain text file called *robots.txt*. It has the access rules to be followed by a well-behaved crawler. This file is written according to the Robot Exclusion Standard[16]. This file contains a list of directories which should not be followed by crawlers and specific crawlers which should not visit the site. For example, consider the following *robots.txt* file:

  User-agent: ArchitextSpider, Lycos_Spider

  Disallow: /cgi-bin/

  Disallow: /Private/

  The file indicates that crawlers *ArchitextSpider and Lycos_Spider* should not visit the site for the purpose of indexing. Directories */cgi-bin and /Private* should not be traversed by crawlers. The Robot Exclusion Standard are for well-behaved crawlers to follow. A server itself can not enforce those restrictions.

- **Robots META Tag** The second convention is at a page level, via the *Robots* META tag placed in the HTML <HEAD> section of a page. Suppose there is a tag in the <HEAD> section:

  <META NAME="ROBOTS" CONTENT="NOINDEX<NOFOLLOW">>

  Then the current page should not be indexed and none of the links in the page should be followed. When parsing the HTML document, crawlers should notice the META tag and check the CONTENT field to recognize that they are not supposed to index the current page or to follow any link in the page. Again, this is not enforced by the server and is only abided by well-behaved crawlers.

2.1.3 Agents

Agents are programs that execute searches. assemble results and present them to the user in a manner tailored to fit a user's profile[15]. One popular application of agents is a meta-search engine. Meta-search engines and other search agents implement basic capabilities of a Web client. They still send normal HTTP requests like any other client except that the requests are only queries on behalf of one or more users and the requests are sent to a collection of origin servers.

A meta-search engine sends a user-specified request on behalf of the user to several search engines. The results are either concatenated or grouped together according to the user's profile. The user can make decisions between various ranking algorithms and capabilities of individual search engines. However. meta-search engines have not achieved the popularity level of ordinary search engines. Users tend to stick to the search engines they are familiar with. rather than sort different results from a few search engines. Furthermore. additional load is imposed on the network in sending requests to several search engines. Longer delays are also caused by the meta-search engine waiting for several search engines to return results. A popular meta-search engine in use today is *metacrawler*. which utilizes results generated by *AltaVista*. *Excite*. *Google*. *Webcrawler*. *etc*.

2.2 Web Servers

A Web server is a program that handles HTTP requests. generates and transmits responses back to clients.

2.2.1 Steps in handling a client request

A Web server usually proceeds through 3 steps to process a request from a client:

- **Convert the requested URL into a file name**

  First the server reads and parses the request headers to extract some header fields such as the requested URL. request method. etc. Then the server translates the URL into the corresponding file name if there is one existing in its file system. Each server has a configurable base directory where Web files are located.

- **Determine whether the request is authorized**

  The server may limit some resources to certain users. To control access to Web resources. the server has an access control list to enumerate the users granted or denied access to the resources. After finding the requested file. the server will check its access control list to determine if the host name or IP address of the client is allowed the access to the file.

- **Generate and transmit a response**

  If the requested file is simply static content. the server invokes system calls to get the file size and the last modification time to put in the response header. together with the response status. the identity of the server and current time. After constructing the response header. the server transmits the header and the contents of the file to the requesting client. For dynamically generated responses. the requested URL corresponds to a script rather than a document. Those URLs typically include a "?" character or a string such as "cgi". "cgi-bin". or "cgibin". If the access permissions assigned to the script permit execution. the script is executed at the server. The server relays input data such as environment variables to the script. In processing the request. the script may contact other servers or interact with its backend database. After the completion of the script. the server receives output and may include some metadata about the resource in the response header before forwarding the response to the client. At some point before or after transmitting the response message. the server may create a log entry in a log file. A response may span several IP packets and would often fit in 10 packets.

2.2.2 Server Architecture

When a Web server handles multiple client requests at the same time. these requests must share access to the processor. disk. memory. and network interface at the server. In terms of allocating system resources among client requests. there are three types of server architectures[15].

Event-driven server architecture

An event-driven server has a process that alternates between handling different requests. The process periodically performs a small amount of work on behalf of each request. The switch from one request to another happens while the server waits for an operation to complete. Servicing different requests through one process allows the server to serialize operations that operate on the same data. which facilitates the sharing of data across different requests. However. the server may take a long time to execute some scripts. This will cause large delays for other requests. Also. alternating between requests requires all the intermediary results for each request to be stored. which introduces complexity into the server software design. As a result. most high-end Web servers do not employ the event-driven model.

Process-driven server architecture

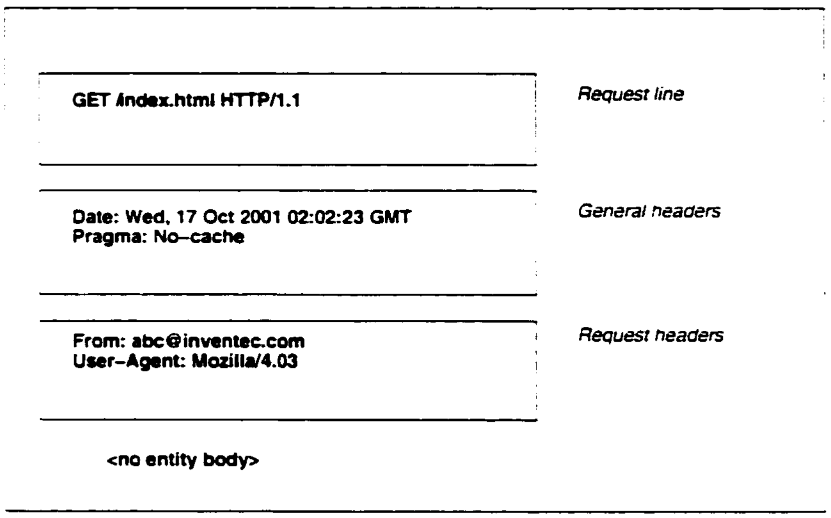
In a process-driven architecture. a separate process is devoted to each request and each process performs all the necessary steps to handle a single request. It depends on the underlying operating system to switch between the various requests. Process-driven servers typically have a master process to listen for new connections from clients. For each new connection. the master process forks a separate process to handle the connection. After parsing the request and transmitting the response. the process terminates. To reduce the overhead of creating and terminating a process for each connection. the master process creates an initial set of processes. Upon the arrival of a new connection. the master process assigns an idle process to the connection. The server terminates a process after it has handled a certain number of requests to protect the system from the gradual consumption of more memory due to memory leaks in the code. The currently popular Unix-based Apache 1.3.3 server follows the process-driven model[20].

Hybrid server architecture

There are three approaches combining the strengths of the event-driven and process-driven models. In the first approach. the process-driven model is generalized so that each process handles a small set of requests. Thus. each process actually becomes an event-driven server that alternates between its small set of requests. The second approach reduces the overhead of switching between processes in the first approach by using a single process that has multiple threads. Each thread is a sequential flow of execution in the shared address space of the containing process. The multi-threaded Web server assigns a thread to each request. The overhead of switching between threads is lower than the overhead of switching between processes because threads share a common address space. The third approach combines the salient features of both event-driven and process-driven models. The event-driven approach is well suited to handling requests that do not perform significant processing or disk access. In this model. the hybrid server uses a main event-driven process to handle the first stages of each request. If the request needs significant computation or a disk access. the main process asks a separate helper process to perform the time-consuming operations. The main process then transmits the completed response to the client.

2.3 HTTP messages

An HTTP message is a sequence of octets that are sent over a transport connection[13]. An HTTP message can be either a request sent from a client to a server or a response from a server to a client.



```
GET /index.html HTTP/1.1

Date: Wed, 17 Oct 2001 02:02:23 GMT
Pragma: No-cache

From: abc@inventec.com
User-Agent: Mozilla/4.03

<no entity body>
```

Figure 2.4: A sample HTTP request

2.3.1 HTTP requests

An HTTP request message, shown in Figure 2.4, has the following syntactic form:

Request-Line

General/Request/Entity Header(s)

CRLF

Optional Message Body

A request message begins with a request line and is followed by a set of optional headers, carriage return-linefeed pair (CRLF), and optional message body. The request line consists of a request method, the URI (Uniform Resource Identifier) being requested, and the protocol version of the client. Optional headers include general headers, request headers and entity headers. General headers can be found both in requests and responses. Request headers only appear in requests and entity headers can be present in requests and responses. In the Figure 2.4, the request method is **GET**, the request source is */index.html*, and the client's protocol is HTTP/1.1. General header *Date* indicates the date and time of the message origination, and *Pragma: No-cache* informs proxies in the path not to return a cached copy. Request header *From* lets the user includes his/her e-mail address as an identification and *User-Agent* is used to include information about the version of the browser software and the client machine's operating system. It has no message body.

Common Request Methods

The HTTP protocol defines a set of extensible request *methods* that are used by a client to perform operations such as requesting, altering, creating, or deleting a Web *resource*. A request method notifies a Web server what kind of action should be performed on the resource identified by the Request-URI(Uniform Resource Identifier). A Request-URI is the URI specified in the request line. A URI is a combination of a URL (Uniform Resource Locator) and a URN(Uniform Resource Name) and can be represented by either or by both. A URN provides a globally unique identifier for a resource, which may reside in one or more locations identified by URLs. For example a resource may be accessible by different protocols such as FTP, HTTP. Therefore, multiple URLs indicate the same resource identified by a globally unique URN. We may have a resource which can be accessed with either FTP URL `ftp://ftp.abc.com/a.html` or HTTP URL `http://www.abc.com/a.html`. Below are some common methods.

- **GET**

  The GET method is the most widely-used method today. It retrieves whatever information is identified by the request-URI. The GET method has no request body. It changes to a *conditional GET* if the request includes an *If-Modified-Since*, *If-Unmodified-Since*, *If-Match*, *If-None-Match*, or *If-Range* header. In that case, the entity will be transferred only if the conditions in the header are met.

- **HEAD**

  The HEAD method is used to fetch metadata about a resource, without a message body being sent. The metadata returned is identical to the information sent in response to a GET request. This method is usually used to verify the validity of cached data. If some new values in fields such as **Content-Length, Content-MD5, ETag or Last-Modified** are different from the old ones, it indicates that the cached data are stale.

- **POST**

  The POST method is used to update an existing resource or provide input to a process. It requests the origin server to accept the data enclosed in the request body as a new subordinate of the resource identified by request-URI in the Request-Line. Its major functions include posting a message to a bulletin board, newsgroup, mailing list and providing a block of data to a data-handling process.

- **PUT**

  The PUT method is similar to POST in the sense that both of them would result in a different version of the resource identified by the request-URI. The request-URI will be updated if there is one existing on the server. Otherwise it will be created. The primary difference between the POST and PUT is the different meanings of Request-URI. The Request-URI in the POST method identifies the resource that will handle the message body enclosed in the request, whereas the Request-URI in a PUT request identifies the enclosed entity in the request.

- **DELETE**

  The DELETE method is used to delete the resource identified by the Request-URI in the request. This method provides a convenient way to delete resources remotely.

2.3.2 HTTP responses

A response message adopts the following syntactic format:

Status-Line

General/Response/Entity Header(s)

CRLF

Optional Message Body

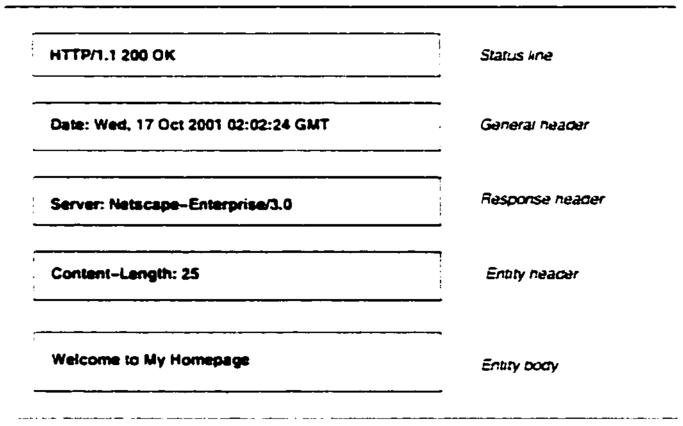


Figure 2.5: A sample HTTP response

A response message starts with a status line. followed by optional General and response headers. carriage return-linefeed pair (CRLF). and an optional message body. Figure 2.5 shows the response message for a GET request. The status line in the response indicates that the protocol the server supports is HTTP/1.1 and the response code is **200 OK**. which means that the request succeeded. The message also includes **Date** General header and **Server** Response header. **Content-Length** indicates the length of the entity body.

2.3.3 Status Code Definitions

Each HTTP response message begins with the *Status-Line*. which include three fields: the server's protocol version number. the response code. and a natural language reason phrase. The server groups various kinds of responses into 5 response classes. each of which has several response codes. Each response code is a three-digit integer.

Success Class Responses

This class of status code informs the client that the request has been accepted. understood or fulfilled.

- **200 OK:** This response is returned if the request has succeeded. The information returned is dependent on the request method. In the case of a **GET** request. an entity corresponding to the requested resource is sent to the client. If a **HEAD** method was used. only the metadata is returned.

- **201 Created:** This response status is returned if the request has been successfully performed and a new resource was created as a result of a **POST** method or a **PUT** method.

- **202 Accepted:** The **201 Accepted** is sent if the request has been accepted and was yet to be completed. Its purpose is to let user's agent continue its task without waiting for the process on the server to be completed.

- **204 No Content:** The server sends **204 No Content** response if the request has been fulfilled and no entity-body is needed to send to the client. For instance. if the user clicks on an inactive part of a imagemap maintained on the origin server. there should be no change to what is displayed to the user. The server then sends back a **204 No Content**. which the browser interprets as no change required.

Redirection Class Responses

This class of status code indicates that further actions need to be taken in order to complete the request.

- **300 Multiple Choices:** Unless a **HEAD** method was used. the **300 Multiple Choices** response includes an entity containing multiple resource locations from which the client can make a choice. If a **HEAD** method was used. no resource locations need to be included in the entity since no response body is returned as a result of a **HEAD** request.

- **301 Moved Permanently:** This response indicates that the requested resource has been assigned a new URI. For a GET or HEAD request. the user agent automatically redirects the request to the new site while an explicit confirmation from the user is required for a POST request. This response is cacheable by default.

- **302 Moved Temporarily:** This response is returned if the requested resource has been moved temporarily. The client should continue to use the old URI. The response is not cacheable unless indicated otherwise.

- **304 Not Modified:** This response is sent if the client has performed a **conditional GET** and the resource has not changed since the last modification time indicated in the request. No entity-body is included in the response.

Client Error Class Responses

This class of status code identifies the errors which are assumed to have been made by the client.

- **400 Bad Request:** This response informs the client that there are syntactical errors in the request and hence it is not recognizable by the server.

- **401 Unauthorized:** The server returns this response to indicate lack of authorization information or the invalidity of authorization information if present in the request.

- **403 Forbidden:** The server understood the request. but refuses to fulfill it. The reason for the refusal may be included in the entity-body if it was not a HEAD request. If it was a HEAD request. no reason will be included in the entity-body.

- **404 Not Found:** The response is returned if the server could not locate anything matching the client's Request-URI.

Server Error Class Responses

The server return this class of status code when it is aware of the errors in itself or can not fulfill the client's request.

- **500 Internal Server Error:** The server encountered an internal error which prevented it from performing the request.

- **501 Not Implemented:** The server returns this response if it does not support the functionality required to fulfill the request. This may happen when the server does not implement the method in the request.

- **502 Bad Gateway:** If a server, acting as a gateway or proxy, received an invalid response from another server, it will send this response to the client indicating it is not the source of the error.

- **503 Service Unavailable:** A **503 Service Unavailable** response is used if the server is temporarily unable to handle the request due to an overloading or maintenance of the server. But this is only a temporary condition and the client may retry the request some time later.

2.4 Pipelining on Persistent Connections

Usually the transfer of HTTP messages depends on the Transmission Control Protocol(TCP) as the underlying transport protocol. However, the setup of a TCP connection involves a three-way handshake and the teardown of a connection requires a four-way handshake. A response of 10 packets follows the connection setup overhead of 3 packets and is followed by connection teardown overhead of 4 packets. Therefore, 7 out of the total 17 packets are overhead. As the Web's popularity increases, more embedded images are included in Web pages. In HTTP/1.0, downloading both the text and images in a page requires multiple TCP connections although some implementations of the protocol include a *Connection: Keep-Alive* request header to request the connection to remain open after the current request. Multiple TCP connections would result in considerable user-perceived latency and extra load on both the server and the Internet. In HTTP/1.1, persistent connections are the default behavior of any HTTP connection. An HTTP/1.1 server or client should assume that the other HTTP/1.1 side intends to maintain a persistent connection unless a *Connection: close* header was sent in the message, which is equivalent to the inclusion of the header *Connection: Keep-Alive* in each HTTP/1.0 message.

A client supporting persistent connections may send several requests without waiting for each response. This is called *pipelining.* The server must send its responses in the order of the requests it received. A comparison of pipelining of requests on a persistent connection and a single request-response exchange on a short-lived connection is illustrated in Figure 2.6. In the example of a pipelining on a persistent connection. the client sends HTTP request 1 following a three-way handshake. Before receiving the response for request 1. the client sends request 2. The server must handle the requests in the order they were received. However. in the second example in Figure 2.6. for every request the client makes with the server. there are three packets required to establish a connection and four packets to close the connection. From this example. we can see that the round-trip delay for each response is substantially reduced by use of pipelining on the persistent connection. Despite the overhead of multiple connection setups and teardowns. many clients. proxies. and servers on the Web are still using HTTP/1.0 and a significant portion of the traffic on the Internet is in HTTP/1.0.

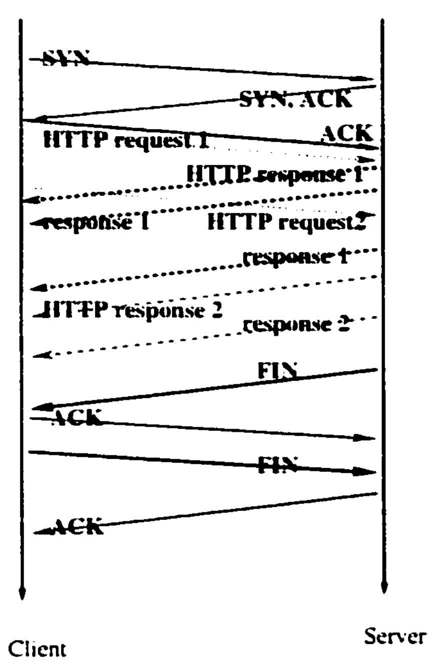
2.5 Web Caching

Web caching is the storage of response messages for later re-use at a point closer to end users in order to reduce response time and network bandwidth consumption[13]. It is probably the most widely studied issue on the Web.

2.5.1 Cache Coherency

A cache may have to ensure a cached response is still fresh before returning it to a requesting client. The origin server can only decide the freshness of a cached response. A cached response is fresh only if it is identical to what is stored on the origin server. otherwise. it is deemed stale. The HTTP/1.1 provides several useful headers for caches to maintain their coherency.

- **Expiration time** The expiration time of a cached response is assigned by the origin server as the time after which the cache must revalidate the response with the origin server before returning it as a response. If the origin server does not explicitly set an expiration time. a cache may



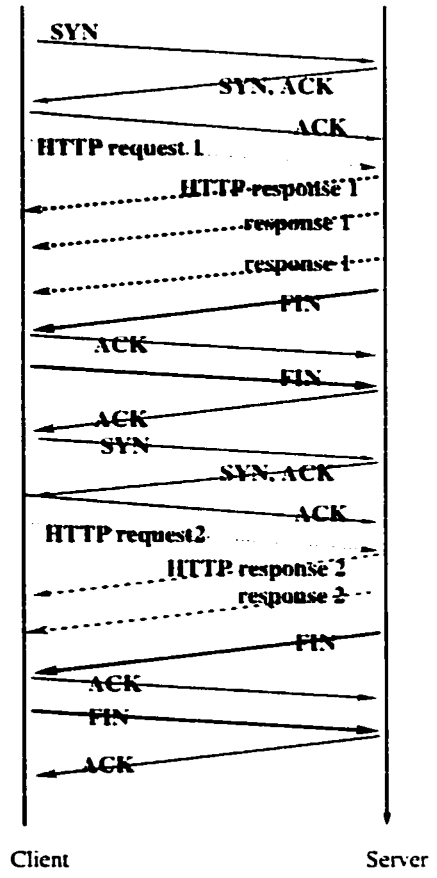


Figure 2.6: Timeline of a pipelining on a persistent connection versus a single request-response exchange on a short-lived connection

assign a heuristic time to the cached response. The heuristic expiration time could be based on the value of *Last-Modified* header associated with the resource. Thus the cache can add a fix amount of time to the *Last-Modified* value to get an expiration time.

- **Last-Modified** A cache can initiate a conditional GET request by putting the value of *Last-Modified* header into *If-Modified-Since* header. If the response has not been changed since then, the server will send back a *304 Not Modified* response without a response body. If the last modified time of the resource is newer than the one specified in the request, the server will usually return *200 OK* with a full response body.

- **ETag** Entity tags (ETag) were introduced in HTTP/1.1 and are primarily used to compare a cached entity against a possible newer version. If a resource has different versions, then each version should have a distinct entity tag. An entity tag is always associated with a specific resource and will never be used to distinguish among different resources. Consequently, for the same resource, if entity tags are identical, then the versions of the resource are the same. Otherwise they are differnt versions of the resource.

2.6 The Application of Crawlers in Indexing

Nowadays, some users begin to surf the Internet through portal sites such as *Yahoo!*, whereas many others use a search engine to start their browsing on the Web. In a search engine, a user typically types in a set of key words, called a query, and a list of pages containing keywords will be returned to the user. Those pages are usually ranked from the most relevant to the least.

Figure 2.7 presents a high level view of a general search engine architecture. Each search either relies on its own crawlers or dedicated crawlers to provide resources for its operation. The *crawler* module is first given a starting set of URLs, and it retrieves their corresponding pages from the Web. The crawlers then extract all the URLs appearing in the retrieved pages following

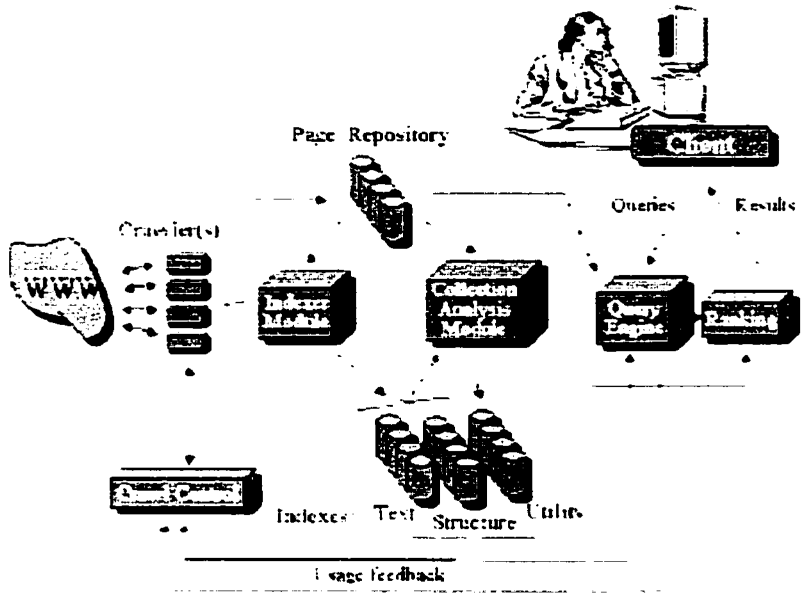


Figure 2.7: General search engine architecture

the breadth-first crawling to find URLs and pass them to the *crawler control* module. This module determines which URLs to visit next. A *page repository* holds the retrieved pages.

The *indexer* module extracts all the meaningful words from each page and records the URL where each word occurs. The result is a huge "lookup table" so that for each word there is a list of URLs where the word appears (the *text index* in Figure 2.7). The *indexer* module also creates a *structure index*, which reflects the links between pages and used to calculate each page's backlink count. The *utility index*, created by the collection analysis module, may provide pages of a given length or pages of a certain importance. The *query* module is responsible for receiving and fulfilling search requests from users while the *ranking* module to filters a significant number of irrelevant pages out of the search results and sorts the results so that results near the top are probably most relevant to user's query.

2.6.1 Crawling the Web

Page Selection

Crawlers start off with an initial set of URLs called seed URLs. The pages those URLs point to are of more general popularity. A search engine might use the *backlink count*. the number of Web URLs that point to a page. to rank user query results. The term *backlink* is used for links that point to a given page. Thus a Web page *P*'s backlinks are the set of all the links on pages other than *P* . which point to *P*. If the crawlers can not visit all pages. then it is better to visit those with a high backlink count. since this will give the end-user higher ranking results.

This type of "citation count" has been applied extensively to evaluate the impact of published papers. Currently. *PageRank* used in the most popular search engine *Google*. extends this idea by not counting links from all pages equally and by normalizing by the number of links on a page[14]. *PageRank* recursively defines the importance of a page to be the weighted sum of the backlinks to it. *PageRank* is defined as below:

- *Assume page A has pages T1...Tn pointing to it. Let C(A) be the number of links going out of pageA. The parameter d is a dumping factor which can be set between 0 and 1. In this case. it is set to 0.85. Then the PageRank of a page A is given by:*
- *PR(A) = (1-d) + d(PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))*
- *The equations can be solved iteratively. starting with all PR values equal to 1. At each step. the new PR(A) value is computed from the old PR(Ti) values until the values converge. The PageRanks form a probability distribution over web pages. so the sum of all web pages' PageRanks will be 1.*

Page Refresh Strategy

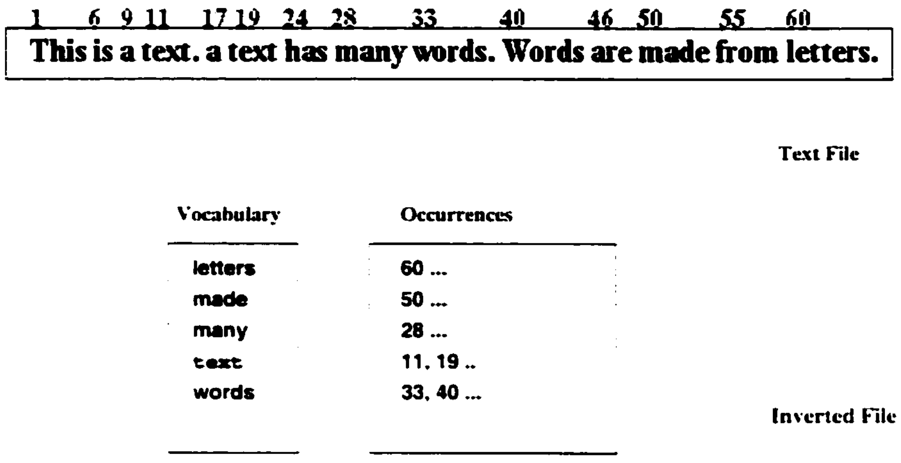
After crawlers have selected downloaded "important" pages. they have to periodically revisit those pages to keep the downloaded pages up-to-date. By up-to-date or fresh. we mean the downloaded pages are identical to their counterparts in real life. Two strategies have been proposed for crawlers to update the pages:

- **Uniform refresh policy:** The crawler revisits all pages at the same frequency. regardless of how often they change.
- **Proportional refresh policy:** The frequency at which the crawler revisits a page is proportional to the change rate of the page. For example. if page *P1* changes 5 times more often than page *P2*. the crawler revisits *P1* 5 times more often than *P2*.

Given limited crawler resources. i.e.. crawlers can only download a limited number of pages within a certain period. which one of these two strategies give us higher "freshness"? We may have an instinct that if a page changes more often. we should update that page more often to keep it up-to-date. However. it has been proven in [2]. that with limited crawler resources. the uniform policy will give us higher freshness then the proportional one for any number of pages. change frequencies. and refresh rate. If there are limited crawler resources. pages that change too frequently should be penalized and not visited very frequently.

2.6.2 Indexing

Among various indexing techniques. *inverted files or inverted indexes* have traditionally been the index structure of choice on the Web. An inverted file(or inverted index) is a word-oriented mechanism for indexing a text collection in order to speed up the searching task. The text collection in our case is a collection of Web pages. The inverted file structure is composed of two elements: the vocabulary and the occurrences. The vocabulary is the set of all different words in the collection. For each word. there is a list of positions or locations where the word appears in the collection. The set of all those lists is called *occurrences or locations*. An example of an inverted index built on a sample text is shown in Figure 2.8. In our example. those positions only refer


1 6 9 11 17 19 24 28 33 40 46 50 55 60
This is a text. a text has many words. Words are made from letters.
Text File
Inverted File

Vocabulary	Occurrences
letters	60 ...
made	50 ...
many	28 ...
text	11, 19 ..
words	33, 40 ...

Figure 2.8: An inverted file built on a sample text

to character positions in a file. For a whole collection of Web pages, a location consists of a page identifier and the position of the word within the page. Sometimes, positions of a word are not tracked and a location only includes a page identifier (optionally followed by a count of the number of occurrences of that word in the page) in order to save space. Given an index term w, and a corresponding location l, the pair $<w, l>$ is referred to as a *posting* for w. At a very high level, building an inverted index over a collection of Web pages involves processing each page to extract postings, sorting the postings first on index terms and then on locations, and finally flushing out the sorted postings as an inverted index over the collection on disk.

The *indexer* in Figure 2.7 builds a text index and a structure(or link index). Using these two indexes and the pages in the repository, the *collection analysis* module builds some other useful indexes.

2.7 Summary

Among the three different forms of a Web client, a browser fetches and renders resources on behalf of users. With the decentralization of the Web, there is no central database of URLs and their associated contents. Crawlers help in gathering pages to assist in searching. Agents execute searches, collate results, and present them in a way tailored to fit a user's profile. In processing a request, a Web server associates the URL in the request with a specific file and decides whether the request is authorized. The server may retrieve the requested file from the disk or invoke a script to generate a response. Although some Web browsers and many Web servers have migrated to HTTP/1.1, HTTP/1.0 is still in wide use today and a significant fraction of the Internet traffic is in HTTP/1.0[15].

Chapter 3

Active Indexing

3.1 Introduction

In this chapter. we first examine the load caused by crawlers on servers. Some experiments were conducted on the logfiles of a department server to substantiate this. Crawlers are used to fetch Web resources to build indexing information for search engines. We developed an efficient mechanism to remove the traffic generated by crawlers from the Internet and at the same time. build the indexing data that search engines need. We propose the use of the active network on which active routers monitor the passing traffic and build indexing data out of the traffic. Then this indexing data is delivered to a dedicated repository via the Internet. After a high level view of our approach is presented about how Web pages are indexed. we explain the architecture of our indexing system in detail. The details include how to process all kinds of IP packets into HTTP messages. which in turn are parsed to generate inverted files (indexing data). and how to send them to the repository. How to organize the distribution of the inverted files in the repository is also described. The gathering of useful global statistics is discussed. At last we illustrate a case study of network programming in PLAN. a Packet Language for Active Networks developed at University of Pennsylvania.

```
crawl4.googlebot.com - - [02/Jan/2002:00:00:34 -0700]
"GET /~yuan/courses/114/99fall/lectures/4syntax/tsld04
7.htm HTTP/1.0" 304 - "-" "Googlebot/2.1 (+http://www.
googlebot.com/bot.html)" "-"
```

Figure 3.1: An example log entry in a log file

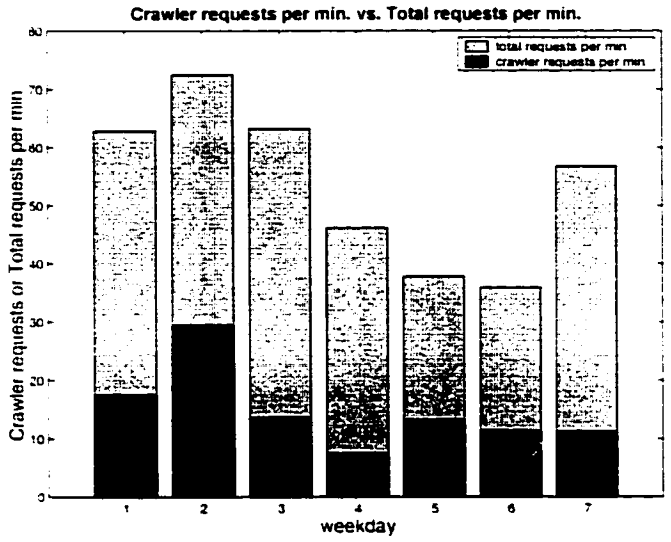


Figure 3.2: Crawler requests and Total requests per min.

3.2 The Load Imposed by Crawlers on Servers

Crawlers play an important role in Web searching, by providing the raw material from which indexes can be built. However, crawlers, like Web browsers, may use multiple connections to read data from the Web server. This can overwhelm servers and force them to respond to crawler requests to the detriment of human visitors[12]. Although some crawlers use multi-threads to open connections with multiple servers simultaneously in order to put less load on servers while Web pages are retrieved at a fast enough pace, crawlers still impose considerable workload on Web servers. Some server administrators claim that up to 50 percent of server hits are from crawlers.

To study the load of crawlers, several experiments were conducted using the log files for the week from June 29, 2001 to July 5, 2001, for the Web server in

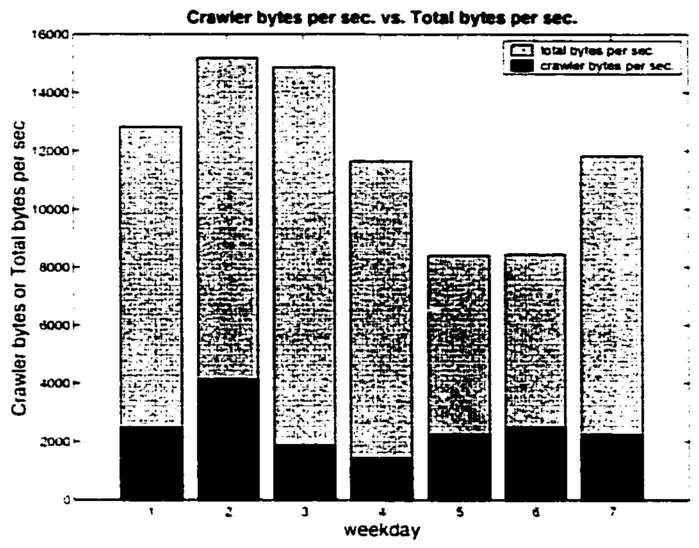


Figure 3.3: Crawler bytes per sec. vs. total bytes per sec.

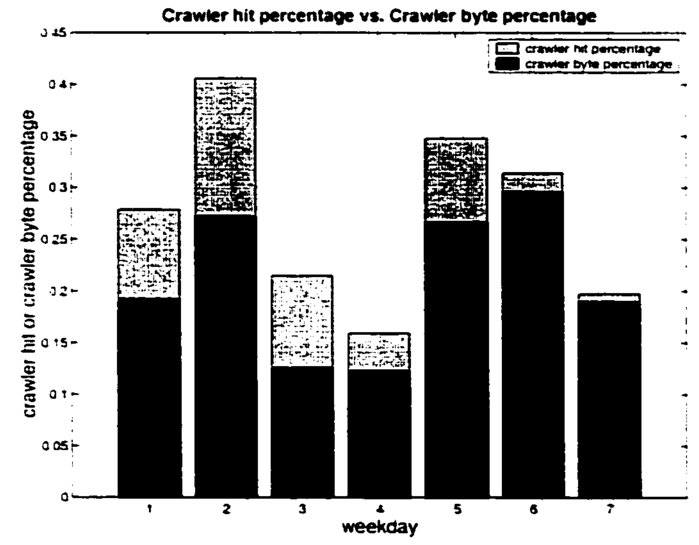


Figure 3.4: Crawler hit and byte percentages

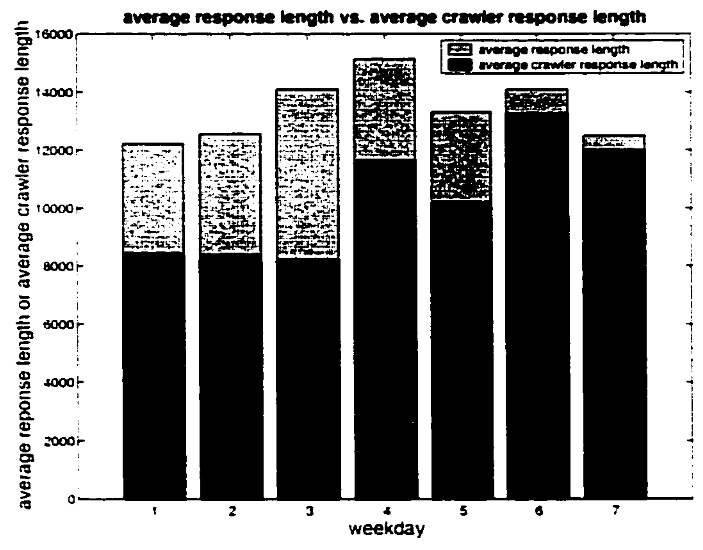


Figure 3.5: Average response length vs. average crawler response length

the Computing Science Department at the University of Alberta. When processing client requests. a Web server generates a log entry in a log file. Each log entry corresponds to an HTTP request processed by the server. including information about the requesting client. the request time. and the request and response messages. An example of a log entry is shown in Figure 3.1. The first field *"crawl4.googlebot.com"* identifies the client's hostname. The server has to perform a DNS lookup to convert the client's IP address gained from the socket associated with the HTTP request into a hostname. If the DNS query fails. the server records the remote host's IP address in this field. The *02/Jan/2002:00:00:34* indicates the time when the server processed the request and the *-0700* indicates that local time is seven hours behind Greenwich Mean Time. The *"GET / yuan/courses/114/99fall/lectures/4syntax/tsld047.htm HTTP/1.0"* indicates the request method. the Request-URI. and the client's protocol version included in the first line of the HTTP request header. The *304* corresponds to the response code included in the HTTP response header. In this case since the response does not include an entity body (*304 Not Modified*). the *content length* field is "-". The *"Googlebot/2.1 (+http://www.googlebot.com/bot.html)"* reports the value of the *User-Agent* request header corresponding to the name and version number of the software responsible for the request.

In order to figure out whether a request was made by a Web crawler or a human client. we utilized the *remote host* field. the first field in a log entry. We gathered the names of Web crawlers available on the Web and built a database of crawlers in the form of either hostname or IP address. Given an entry in a log file. if its *remote host* field can be found in the database. we determine the request was initiated by a crawler. otherwise it is from a human visitor.

Figure 3.2 shows the number of requests per minute caused by crawlers and the number of total requests per minute. Total requests include the requests initiated by both a variety of clients and crawlers. The number of requests from crawlers hits a maximum of 30 per minute with the average over the week of around 16 per minute. whereas the average of total requests per minute is 50. Figure 3.3 demonstrates that the number of total bytes per second the server transmitted in response to all kinds of requests and the number of the bytes transmitted by the server in handling requests from crawlers. It indicates that the server sent an average of about 2500 bytes per second to crawlers. Figure 3.4 illustrates the ratio of crawler hits to the total hits caused by both human visitors and crawlers and the ratio of bytes fetched by crawlers to the total bytes for the week. The average hit and byte percentages are 27.3% and 20.9% respectively. The hit percentage has a maximum of 40.6%. The byte percentage reaches 29.5%. and the byte percentage is always smaller than the hit percentage. The reasons are explained in the next paragraph.

Figure 3.5 shows the comparison between the average response length (in bits) and the average crawler response length. From the graph. we can clearly see that the average response length caused by crawlers is always less than the overall average response length. There are two reasons for this. First. crawlers only fetch standard HTML pages and ignore all other media and document types. such as PDFs. images. and sounds. which are usually larger than standard HTML pages. Secondly. crawlers from some services use the HTTP HEAD command to get meta information about a page before actually fetching it. In such a case. the crawler will only fetch pages that have been changed.

Due to the incompleteness of our database of Web crawlers. there could be logs on our server that were caused by crawlers. but were not recognized as such. Also. the server of our department only hosts a subsidiary site (*www.cs.ualberta.ca*) of a main site (*www.ualberta.ca*). Many more requests by crawlers would go to the main site as crawlers usually start off with "important" URLs. The site *www.ualberta.ca* is obviously more "important" than *www.cs.ualberta.ca*. Considering the above two factors. the real workload imposed by crawlers on servers would likely be more than what we measured on our server.

3.3 Our Approach

3.3.1 Introduction

The load imposed on Web servers will also finally be put on the network since it is the underlying network that is responsible for transmitting responses generated by servers to clients. In this chapter. we propose the use of a novel network architecture - an active network - to remove the need for crawler traffic from the Internet. and incidentally from the Web servers themselves. In an active network. the routers or switches of the network can perform customized computations on the messages flowing through them. These networks are active in the sense that nodes can perform computations on. and modify the contents of. the packets flowing through them[17]. In traditional networks. user programs execute only at the end systems and the network passively transports user data between these end systems. In active networks. users can supply both data and code to the network. and the network may execute user code on one or more intermediate nodes while transporting packets between endpoints.

Our proposal is to install a packet monitor and indexer on active routers. These modules would be distributed to the routers using the capabilities of the active network. Once installed. the monitor looks for all Web pages passing by. The indexer then processes those Web pages out of a bunch of *ip* packets to produce an inverted index. The inverted index is compressed. using an efficient storage scheme such as the mixed-list scheme. The compressed inverted index

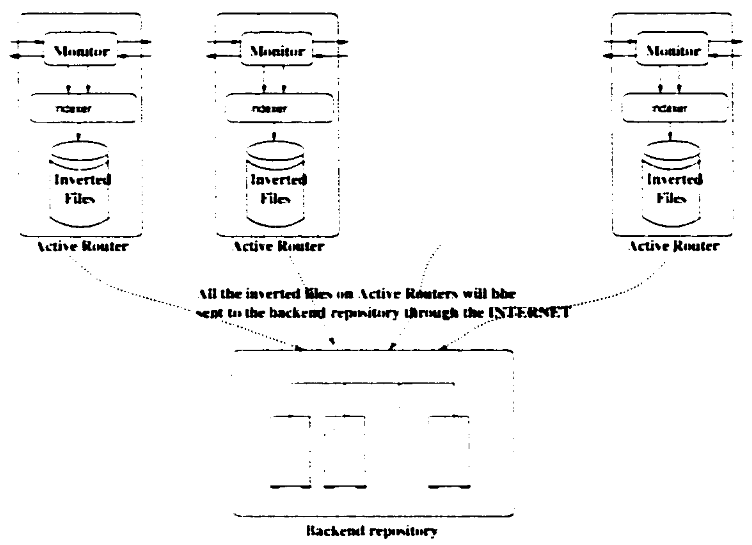


Figure 3.6: high level architecture of the indexing system

is then transferred to a dedicated backend repository via the Internet and merged there with inverted indices from other routers. We present a high level architecture of the indexing system in Figure 3.6. This approach would generate traffic from the active routers to the repository in the form of indexing information for pages fetched by normal clients.

There are some issues with the current implementation of crawlers. Following analysis in [23], the Web, represented as a graph, can be divided into five main components: a central strongly connected core*(SCC)*, *OUT* consisting of a set of pages that can be reached from the core but do not connect back to it, *IN* consisting of a set of pages that can reach the core but cannot be reached from it, *TENDRILS* containing pages that cannot reach the core and cannot be reached from the core and disconnected components as shown in Figure 3.7. A crawler typically starts with a base list of popular sites known as *seed URLs* and follows links within the sites[14]. The pages in *SCC* are relatively popular ones since all of them can reach one another along directed links. Therefore, crawlers are more likely to start off with nodes in *SCC*. According to the definition of each component, the nodes in *SCC* and *OUT* can be easily captured. But it is hard for those nodes in *IN* and *TENDRILS* to

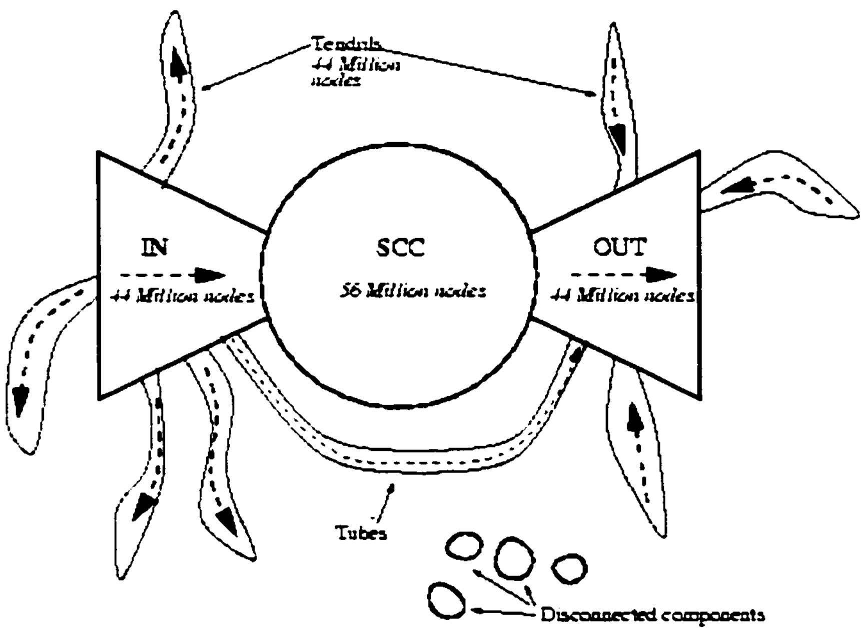


Figure 3.7: Connectivity of the Web represented as a Graph *([23])*

be visited and even harder for those in *DISCONNECTED* to be touched as a result of the unique characteristic of the Web graph. Pages in *IN*, *TENDRILS* and *DISCONNECTED* are not reachable from *SCC* and whose links do or do not exist somewhere on the Web. They are "hidden" from crawlers. Secondly, current crawlers cannot generate queries or fill out forms, so they cannot visit "dynamic" contents. This problem will get worse over time, as more and more sites generate their Web pages dynamically from databases. However, these problems can be fixed via active indexing since active routers will index whatever pages pass by, including static and dynamic pages as well as "hidden" pages. In this sense, the indexes generated this way will be more complete than those generated via crawlers because the Web pages which cannot be reached from the seed URLs or by links in the seed pages will probably never be indexed. Furthermore, if all crawlers can be removed, the total load on both the Web servers and the Internet will be reduced considerably.

Active indexing is also capable of providing a "dynamic" view of the Web, by building up statistics on dynamic behavior such as the number of times per hour a particular page has been fetched. These statistics could be used during index generation in combination with other common indexing factors such as the *inverse document frequency* and *citation count*. In contrast. crawling the Web gives us only a static picture of its contents. Statistics such as the number of times a particular page is linked to by other pages can be used to weight the importance of a page. but that is only a proxy for how many times that particular page is actually fetched. Thus. active indexing gives us not only the static statistics about a Web page. such as the number of times it is linked to by other pages. but also the dynamic statistics including the number of times per unit time it is seen on a router. Consequently. these features of a page can be combined to generate a query result. which tends to be more accurate.

In our scheme. active routers passively wait for traffic to come. For those recently updated Web pages or new pages which have not been visited by human visitors. our active routers are not able to capture them. The index data of those pages which have been indexed before would have the potential to be out-of-date and the index data of new pages might not be existing in the database. We will further address this issue in our future work in Section 5.2.

3.3.2 Placement of active routers

The Internet can be viewed as a collection of subnetworks connected by backbone routers. For each subnetwork. there is a gateway router to connect it to a backbone router. Usually there will be more than one intermediate router between a client and a Web server. However. each packet should only be indexed once on the way to its destination. One of the following 3 strategies could be used in deciding where to place the active indexing:

- **trace HTTP traffic on backbone routers**

  The first option is to monitor HTTP traffic on backbone core routers. The core routers between the client and server see both the HTTP requests and responses. Thus most HTTP traffic traveling through the network will be indexed - only those connections where both the server and client are in the same stub network will be missed. This scheme has several disadvantages. Firstly. if there is more than one core router between the client and server. each page will be indexed multiple times. Secondly. even if each page fetched is only indexed once. it is very likely that a particular Web page will be fetched via multiple backbone routers. since Web sites are usually visited by users all over the world. As a consequence. several core routers would all index the same page. Sending the inverted index for each replica to the backend repository will waste bandwidth. Thirdly. core routers are heavily taxed just in their normal role of forwarding packets: any additional CPU load on these machines is very unwelcome.

- **trace HTTP traffic on routers one hop away from servers**

  Due to asymmetric routing in the Internet. it is very unlikely that a router will see both an HTTP request and its matching response unless the router is exactly one hop from the Web client or Web server. To avoid congestion on busy links. we could select one route randomly from among the first few shortest paths to the closest backbone router. The node one hop away from the server on that route would be selected as the monitoring node. The drawbacks of this scheme are that there will be as many monitoring nodes as there are servers. and that not all traffic from a particular server will be seen.

- **trace HTTP traffic on gateway routers**

  To overcome the drawbacks of the first two schemes. we can index Web pages on the gateway routers of the stub networks connected to the network core. Since there will be more than one server on each stub network. there will be fewer monitoring nodes than servers. The gateway routers. by definition. see all traffic between their network and the rest of the world. so all traffic will be indexed. Additional processing load on the gateways will be much more easily accommodated than on the core routers. Duplicate indexing is ameliorated by having each gateway node only take charge of monitoring the Web servers in its adjacent stub

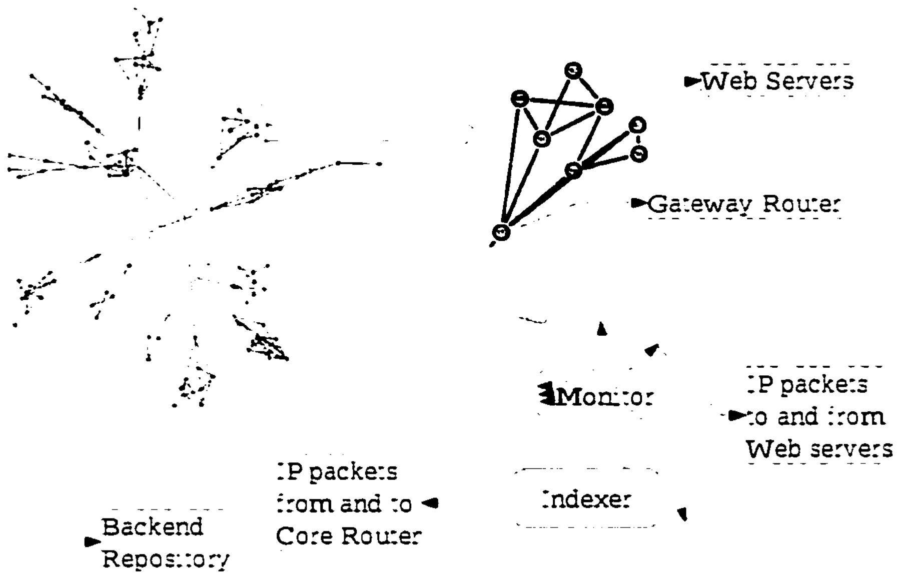

Web Servers
Gateway Router
Monitor
IP packets to and from Web servers
IP packets from and to Core Router
Indexer
Backend Repository

Figure 3.8: overview of indexing system

network. However. compared to indexing on core routers. there will be many more indexing nodes. We deploy this strategy due to its flexibility and the benefits we will gain.

3.3.3 Architecture of Our Indexing System

Our system for building inverted indexes is based on the Internet model. All the gateway routers compose a distributed shared-nothing architecture. Figure 3.8 demonstrates the indexing system from the view of the Internet. We recognize the following three types of nodes in our indexing system(as shown in Figure 3.8):

- **Web servers** These nodes store a collection of resources to be indexed. If some resource on a server is fetched from outside and passes through the server's gateway router. it will be indexed on the gateway router.
- **Gateway routers** Each gateway is installed with two processes - packet monitor and indexer. The packet monitor watches all the traffic passing

by. If the Web traffic is of interest. then it will be saved and indexed later on.

- **Backend Repository** The inverted indexes built on gateway routers are sent to the backend repository and merged there with indexing data from other routers.

The input to the packet monitor on a gateway is a sequence of IP packets of all types. They could contain any type of TCP packets or UDP datagrams. The output consists of filtered IP packets which are only the enclosed HTTP message packets. which in turn. are the input to the indexer. The output of the indexer is the compressed data of the inverted file. The inverted file covers the documents on the servers in the stub network. which have been seen on the gateway. A resource on a Web server will be indexed if and only if it is requested and passes through the server's gateway. The monitoring process at the gateway monitors all the passing traffic. but only keeps HTTP requests headed for the servers in its stub network and HTTP responses originating from the servers. The monitor filters out any other traffic including HTTP request packets from within its stub network and HTTP response packet from outside. What the monitor receives are IP packets which may or may not be complete Web pages. which are what crawlers fetch. These IP packets are demultiplexed into integral pages. These pages are processed to generate inverted indexes. Thus we require gateway routers to have the following capabilities:

- Each gateway knows the IP addresses of all the servers in its stub network.

- The packet monitor on a gateway is capable of distinguishing between a HTTP request packet and a HTTP response packet. Together with the knowledge of its servers. the gateway is able to tell if the current packet is a request packet destined for its stub network or a response from a server inside the stub network. All the packets of other types are filtered out and dropped. To monitor HTTP traffic. the packet monitor can consider traffic traveling to or from the well-known port 80. although some Web sites choose to use an unreserved port such as 8000 or 8080 instead. In order to distinguish between requests and responses. we will parse the IP packets and extract their IP and TCP headers. The source IP address and port number in the request will appear as the destination IP address and port number in the response. We can use this feature to find a pair of communication parties: the client and the server. Then. given all the request packets and the response packets between the two communication parties. we should associate each request with its corresponding response. Since the HTTP response is the first data that the server sends back to the client and will acknowledge the last byte of the HTTP request. the first sequence number of the HTTP response should be equal to the acknowledgment number of the request. Consequently. the match can be done based on a match between sequence numbers and acknowledgment numbers.

- The indexer divides its procedure into two stages. In the first stage. individual IP packets are processed to reconstruct HTTP messages. The reconstruction of HTTP messages involves handling out-of-order. corrupted. and duplicated packets. First we may demultiplex packets according to their source and destination IP address. and then reorder packets according to TCP sequence numbers and eliminate duplicates[18]. With all the HTTP requests and responses. we need to associate each response with its corresponding request since the URL is not present in response headers. HTTP requests are matched with responses by comparing sequence numbers and acknowledgment numbers. In the second stage. for all URLs produced in the first stage. a hash table is used to hash URLs into page identifiers for the convenience of quick location of a URL. Then each page is parsed to remove HTML tagging. tokenized into individual terms. postings(term-location pairs) are extracted to build an inverted file. The inverted file is compressed using a so far most efficient compression scheme. called mixed-list storage scheme[21]. We will explore this compression scheme further in Section 3.5 in this chapter.

The indexer builds an inverted file of all the index terms along with their associated statistics. The inverted index is compressed using the mixed-list storage scheme. Then the compressed data is shipped to a backend repository via the Internet.

3.3.4 Distributed inverted index organization

In a distributed environment, there are basically two strategies for partitioning the inverted index across a collection node in a repository.

The *local inverted file* [22] organization partitions the document collection so that each query server stores a disjoint subset of documents in the collection. A query server or query engine receives and fulfills user queries. A search query would be broadcast to all the query servers, each of which would return disjoint lists of document identifiers containing the search terms.

The *global inverted file* [22] partitions based on index terms so that a query server stores inverted lists only for a subset of the index terms in the collection. For example, in a system with two query servers A and B, A could store the inverted lists for all index terms that begin with characters in the range *[a-q]* whereas B could host the inverted lists for the remaining index terms. Therefore, a query that asks for document containing the term "*procedure*" would only involve A.

In the global inverted file organization, when a query server that stores some set of index terms fails, search queries dealing with that set of index terms can not be answered. On the other hand, a similar failure on a single query server in a local inverted file organization does not prevent most search queries from being answered, though the results might not contain all the relevant documents in the collection. Performance studies in [19] also indicate that this organization processes search queries effectively and provide good query throughput in most cases. Considering the above analysis, we can deploy the *local inverted file* organization in our backend repository. The repository consists of a collection of nodes connected by a local area network. We distribute the inverted index across different nodes so that each node is responsible for a disjoint subset of documents. Each node receives inverted indexes from a small disjoint set of gateways. Since each gateway only indexes sources on a disjoint subset of servers. each node in the repository save inverted indexes for a mutually disjoint sub collection of documents.

3.3.5 Encoding

Before being sent to the backend repository. the inverted files are compressed in order to minimize the volume of transferred data. thus make better use of the network bandwidth. Inverted files can either be stored as *Full list* or *Single payload* as sets of *(key. value)* pairs. In the *Full list* scheme. the *key* is an index term. and the *value* is the complete inverted list for that term. Since an inverted file is a text index composed of a vocabulary and a list of occurrences. it is the most natural way to store an inverted file according to the definition. On the other hand. inverted files are also quite amenable to compression. This is because the lists of occurrences or inverted list are in increasing order of text position. Therefore. an obvious choice is to represent the differences between the previous position and the current one. In [21]. a *Mixed list* scheme was proposed. In this scheme. the key is a posting. i.e.. an index term and a location. The value contains a number of successive postings in sorted order. including different index terms. The postings in the value field are compressed and in every value field. the number of postings is chosen to make the length of the field approximately the same. For those terms which have a long inverted lists. it is possible to have their inverted list be spread across multiple *(key. value)* pairs.

Figure 3.9 illustrates the full list scheme and the mixed-list scheme. The top half of the figure depicts the full list for four consecutive words and the bottom half shows how they are stored in the mixed-list scheme. As indicated in the figure. the *keys* part contains the postings (low. 200). (low. 349). (lowliness. 252) and the postings in the *values* part are compressed by using prefix compression for the index terms and by representing successive location identifiers in terms of their numerical difference. For example. the posting (lower. 235) is represented by the sequence of entries *3 er 235*. where *3* is the length of the common prefix between *low* and *lower*. *er* is the remaining suffix for

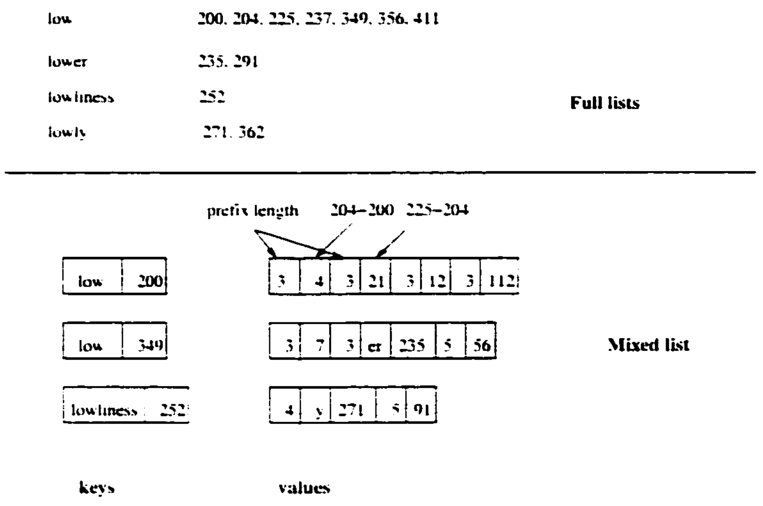


Figure 3.9: Mixed list storage encoding scheme vs. Full list encoding scheme

lower, and *235* is the location.

Studies in [21] indicate that the mixed-list storage scheme scales very well to large collections. The resulting index size is consistently about 7% the size of the input HTML text. This is so far reported the most efficient compression scheme for inverted files. So if the indexer on a gateway builds indexing data on all the input it receives and the generated inverted index is compressed by using the mixed-list scheme, the final compressed data which will be sent to the backend repository is roughly 7% the size of the original HTML text.

However, although many resources on the Web change rapidly, chances are that a resource just indexed on a gateway is still up-to-date when the same resource passes through the gateway the next time. [3] studied the average change interval of a page. Figure 3.10 shows the statistics collected over all domains. In the figure, the horizontal axis represents the average change interval of pages, and the vertical axis shows the fraction of pages changed at the given average interval. For instance, for the third bar we can see that 15% of the pages have a change interval longer than a week and shorter than a month. If we assume that the pages in the first bar change every day and the pages in the fifth bar change every year, the overall average change interval of a Web page is about 4 months. The changes of Web pages follow a *Poisson*

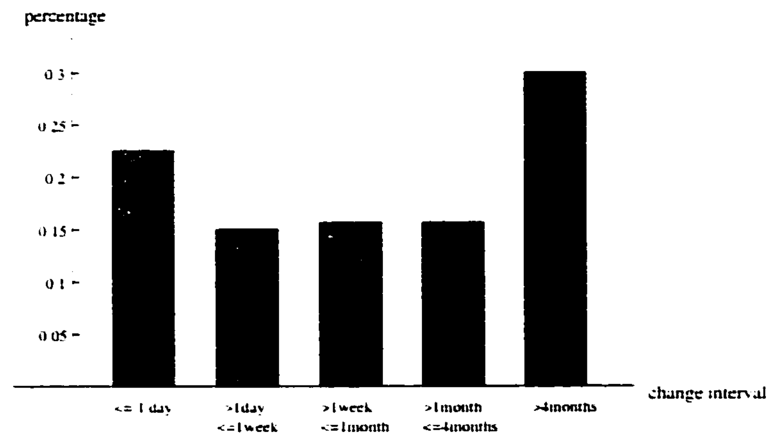


Figure 3.10: Fraction of pages with given average interval of change

process[3]. A Poisson process is usually used to model a sequence of *random* events that happen independently with fixed rate over time.

We only need to reindex stale pages when they are seen again on gateways and therefore, do not need to reindex all the information fetched by indexers on gateways in order to use the indexing system efficiently. To implement this, a hash table is used on each gateway to associate a URL with a page identifier. Page identifiers of all the pages seen by a gateway are saved in the hashtable. For each identifier, we store the value of the *Last-Modified* field. If a document was previously fetched by the gateway and the *Last-Modified* header in the more recently fetched file is newer than the old value, the file is reindexed and the *Last-Modified* in the hashtable is updated. If it has been indexed, and the two *Last-Modified* values match, the file is dropped and not reindexed. If it has never been seen by the gateway, a new entry is created in the hashtable. Besides the *Last-Modified* header, other headers including *Expires* and *ETag* can also be used to compare an old document against a possibly newer one. By not indexing all the files gathered by the indexer, the final index size sent to the repository should be far below 7% of the size of the original HTML text and considerable system resources and network bandwidth can be saved.

By building distributed indexing data across gateway nodes and forwarding them into the backend, we smoothly remove the need for crawlers and thus remove the traffic caused by crawlers from the Internet. By this means, we believe the extra traffic in terms of bytes imposed on the network is up to 7% in the worst case, which compared to the average extra workload of 25% from crawlers is trivial.

3.3.6 Communication Mode between Gateways and the Backend

The partial indexing results, which are distributed on gateways, are delivered to the backend repository and assembled to generate the inverted index over the whole collection. There are two options of the delivery mode.

Dribble Mode

In the dribble mode, the indexing traffic is constantly sent to the repository. Whenever a gateway sees a fresher page, the inverted index for the page will be delivered to the repository almost immediately. In this way information in the repository is maintained up-to-date to the maximum. But this mode does not take into consideration the current condition of the network. If the network is already overloaded, the extra indexing traffic will deteriorate the situation. In our simulation, we use dribble mode to forward the indexing data to the repository.

Batch Mode

In batch mode, a batch of indexing traffic is forwarded to the repository once in a while. This mode takes into account the current network situation. A gateway will not transmit any indexing traffic until it observes the traffic is not heavy in the network. But in the worst case, if the network has been busy for a long time and the accumulated indexing results have been kept in gateways, many pages in the repository will become out-of-date.

3.3.7 Gathering Statistics

In our indexing system, we gather two important global statistical metrics: inverse document frequency for indexing terms and document visiting frequency for documents.

Inverse Document Frequency

Given a specific term or word, the *inverse document frequency* or *idf* is the reciprocal of the number of documents where the term appears. The motivation for usage of an *idf* factor is that terms which appear in many documents are not very useful for distinguishing a relevant document from a non-relevant one. Used in ranking the query results, it helps return documents near the top which tend to be relevant to a user's query. To gain the *idf* factor for a particular term, we have to get the document frequency first.

Only inverted indexes are produced on gateways to save system resources and network bandwidth. However, besides inverted indexes, we also need to create a lexicon that lexicon lists all the index terms in the inverted file along with their associated statistics in the repository. The associated statistic, in our case, is document frequency in the repository. Each node in the repository stores inverted files, a subset of documents in the collection. The inverted files only tell local information about document frequency for a term. A dedicated statistician is needed to aggregate all local information. Figure 3.11 illustrates how document frequency statistics are collected for each term. In Figure 3.11, the statistician summarizes and merges inverted lists from different nodes. The output is a sorted stream of global document frequencies for each term which is sent back to each node. Only statistics about terms present at a local node are sent back to that node. In the example, statistics about "raspberry" are not sent back to node 1 since the term is not present in the local collection.

Document Visiting Frequency

Document Visiting Frequency (dvf) is the number of times per unit time a page has been seen by the monitoring process on a gateway. It gives us a dynamic view of the Web. As opposed to the static statistics *citation count* used by some search engine currently, *dvf* factor, when used in ranking documents, tends to result in a higher rank for a user's desired results. We gather this data on gateways. In the hashtable associating a URL with a page identifier,

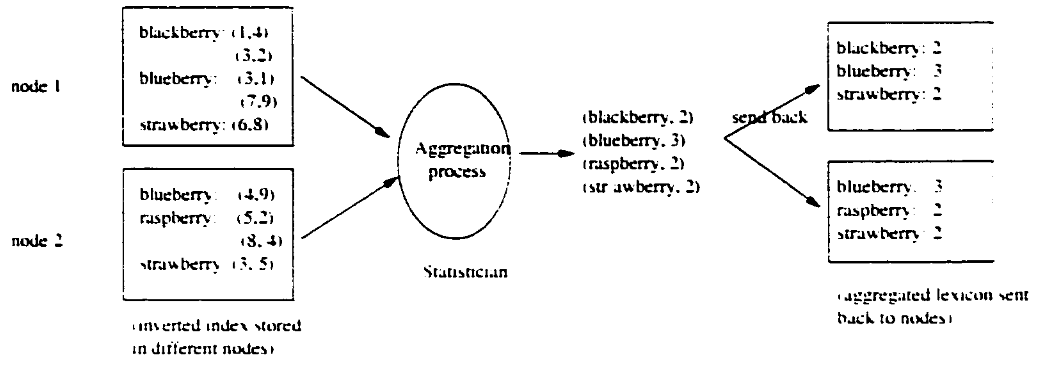


Figure 3.11: Collect document frequency

besides the *last-modified* field. a *dvf* field is also added. For a document fetched by the indexer. whether it is fresh or stale if previously indexed. the *dvf* counter for the document is always increased by 1. These data are periodically transferred to the repository to be used in generating queries. If two documents contain a query keyword. the one with the bigger *dvf* is ranked higher. It also can be combined with with *idf* factor in answering a search query. Due to strategic locations of gateways. a gateway sees all the traffic between its adjacent stub-network and the rest of the world. For a Web resource located within a stub-network. its gateway is always able to capture the resource as long as it is visited by clients from outside of the stub-network. although other gateway routers may also see the resource. In this sense. the *dvf* counter for a document is global.

3.4 A Case Study of Network Programming Using PLAN

A number of research groups have been actively designing and developing active network prototypes. Among them. a new Packet Language for Active Networks(PLAN). developed at University of Pennsylvania. is a scripting language whose programs are intended to be carried in packets and executed on routers or hosts through which the packets pass[9]. PLAN was attempted to fit within an architecture that balances flexibility. safety. security. and performance. For example. all PLAN programs must terminate in order to protect the network while making it available to users, and may not access information private to the router or other programs.

PLANet, an internetwork, uses active packets written in PLAN and implements network layer services directly on top of the link layer, independent of the existing IP infrastructure[10]. A PLAN program may invoke node-resident services written in OCaml[11], a dialect of the ML(MetaLanguage) programming language. These services are dynamically loaded into a PLANet node and only implement essential functionality to operate the network, like address resolution, routing, DNS, etc. Therefore, PLAN packets, coupled with general-purpose services form a two-level active network architecture for PLANet.

We built a small application in PLAN. An active packet contains a destination address. When an active packet passes through an intermediate router on the way to its destination, a key is generated to identify individual packets, and then the key is saved on the node together with the destination address. To implement this, a small scale active network was constructed, and on each node of the network, an empty hash table was allocated. The code segment in Ocaml for the definitions and allocation of the hashtable is shown as follows:

```
(* Configuration constants *)
let index_store_size = 200 (* max num packets to keep track of *)
```

Define the hashtable:

```
type index_info =
 { mutable keynumber:  Activehost.activehost * int; (*packet id*)
 mutable dests:  Activehost.activehost (* destination host *)
 rwl :  Rwlock.t }
```

Allocate a hashtable with size 200 on each node:

```
let itbl:  (Activehost.activehost, index_info) Hashtbl.t = Hashtbl.create
index_store_size
let itbl_rwl:  Rwlock.t = Rwlock.create() (* lock for the hash
table itself *)
```

When an active packet goes through an intermediate node, the packet will be saved in the hashtable on the node provided that no packet with the same destination address has been stored on the node. A read/write lock is also created at the same time in order for the hash table not to be accessed by other processes. The code segment is shown as below:

```
let savehsh(mykey, dest) =
 try(
 with_read itbl_rwl
 (function _ -> Hashtbl.find itbl dest) (); false)
 with Not_found ->(
 with_write itbl_rwl
 (function _ ->
 Hashtbl.add itbl dest {keynumber = mykey;
 dests = dest; rwl = Rwlock.create()}
 ) ();
 true)
```

After we map the string. "saveDestHsh". defining the name of the PLAN service to the function that implements that service. savehsh(). the services must be installed in the PLAN router. The program file of mapping service names to functions is included in Appendix B. We then can inject into a specified active router an active packet carrying programs written in PLAN that invoke the router-resident service we implemented. An example of PLAN packet that invokes the service "saveDestHsh" is shown in Appendix C.

In addition to the service of saving data into the hash table on an active router. we also implemented other services including retrieving the data from the hash table. deleting some specific entry and empty the hash table. We include all the Ocaml code that implements all these services in Appendix A.

According to our experience. Ocaml. as a functional language. supports programming in a functional style and emphasizes the evaluation of expressions. It works with the evaluation of *expressions* rather than *statements* which are used in imperative languages such as **C**. **C++** and **Java**. It is all about *expression* since everything reduces to expression. In an extreme case. one program could be one expression. Functions are often passed around within a program in much the same way as other variables. Recursions are heavily employed in Ocaml as a primary control structure. as opposed to *loops* in imperative languages. The resultant codes are often much shorter and reliable than their equivalent imperative codes. However. it requires a fairly abstract thinking style. heavily influenced by mathematical principles and codes are often far from readable to a layman programmer. All these features make Ocaml more like a research language and prevent it from being widely used in industry.

3.5 Summary

The experiment results on the logfiles of a department Web server gives us a vivid picture of the workload imposed by crawlers on the Web servers and the Internet. Then we give a detailed description of our indexing system. Unlike search engines contingent on crawlers actively retrieving Web pages. our indexing system accesses Web pages by passive monitoring in order not to put extra load on the Internet. Due to the unique feature of the graph structure in the Web. there are always pages unspotted by crawlers if they fall into components such as disconnected components. Our system is capable of gathering more complete statistics by allocating monitoring nodes in all stategic places on the Internet. In addition to static statistics like the citation count of each page and Inverse Document Frequency. our scheme also can gather dynamic statistical data about a page such as global Document Visiting Frequency. Both static and dynamic statistics of a page can be combined by search engines to answer a user query.

Chapter 4

Simulation Implementation

4.1 Introduction

Simulation is used to evaluate the performance of computer systems and provides an easy way to predict the performance or compare several alternatives[4]. This chapter begins with the methodology we used in our simulation. One of the greatest challenges in conducting computer simulations is identifying and measuring correctly the metrics of interest. That involves the removal of the transient state and the best estimation of our system's response in the steady state. The simulation implementation is then discussed including network configuration. simulation parameters as well as performance metrics. The chapter is concluded with our experimental results and analysis.

4.2 Simulation Methodology

4.2.1 Simulation Model

We use the discrete-event simulation model. which is used in computer systems since the state of the system is described by the number of jobs arriving at different devices. Our experiments were conducted on SMURPH[1]. a simulator oriented toward investigating medium access control(MAC) level protocols in communication networks. It hides the simulation details from the user. All simulation-related operations like creating and scheduling individual events and maintaining a consistent notion of time. are covered by a high-level interface presented as a collection of abstract objects. methods. and functions.

A program of a protocol description in SMURPH is executed on hypothetical communication hardware. The user extends some standard data types and interrupt-driven processes structured like *finite-state machines* to build the program of a specific protocol. Since we do not counter the increasing processing time for increasing queue length, we allow infinite queue in our simulation.

4.2.2 Transient Removal

The key to conducting proper computer simulations is the deceptively simple task of determining when to actually start taking measurements. Typically, a computer simulation does not immediately reach its equilibrium point (or steady state), but only after a certain amount of simulation time has elapsed. The response of our system prior to equilibrium corresponds to the transient part of our simulation. It is important to employ proper simulation techniques to correctly identify and phase out the effect of the transient part (transient removal) and measure metrics only during the steady state.

A characteristic of the steady state is the fact that the response of our system exhibits very low variance when equilibrium is reached[4]. In other words, the rate of change of our metrics approaches zero. In our simulations we employ the throughput of our system to determine steady state. At fixed time intervals we obtain a sample of the system's throughput, $throughput_i$. In order to ascertain whether our system has reached equilibrium or not, we utilize three consecutive samples of the system throughput and calculate their relative differences as follows:

$$difference1 = \frac{throughput_{i-1} - throughput_{i-2}}{throughput_{i-2}} \tag{4.1}$$

$$difference2 = \frac{throughput_i - throughput_{i-1}}{throughput_{i-1}} \tag{4.2}$$

If both *difference1* and *difference2* are less than a pre-specified value, which we chose to be 1%, we then determine that the simulation has entered the steady state. We follow the aforementioned transient removal procedure for all different network loads. Thus, we make sure our system is in steady state

Request Inter-Arrival Time(tick)	Steady State Point(tick)
12000	2.46e+11
15000	2.55e+11
18000	3.58e+11
19000	3.62e+11
20000	3.21e+11
21000	1.26e+11
22000	0.74e+11
23000	0.85e+11
27000	0.81e+11
30000	0.91e+11
40000	0.96e+11
50000	0.92e+11
100000	1.29e+11
200000	0.49e+11

Table 4.1: Different Steady State Points

before we start monitoring and fetching the passing HTTP messages to generate indexing information for those HTTP messages. Figure 4.1 shows the response of our system versus simulation time for different network loads. Table 4.1 illustrates different simulation times when simulation cases reach their corresponding steady state. From both the table and the figure. it is evident that at time *4e+11*. all the simulations cases have reached steady state. Therefore. we start accumulating indexing traffic at time *4e+11*.

4.2.3 Variance Estimation

The output of our system at steady state is dependent upon the initial seeds for the pseudo-random number generator. In order to minimize the effect of our initial seed selection and obtain a relatively accurate estimate of the system's response. we repeat each experiment several times using several sets of seeds. For each case. all replications of an experiment consist of a complete run of the simulation with the same set of input parameter values. The replications differ only in the seed values used for the random-number generators. Our method consists of conducting 8 replications for each case. However. it is not possible to get a perfect estimate from a limited number of samples. A confidence

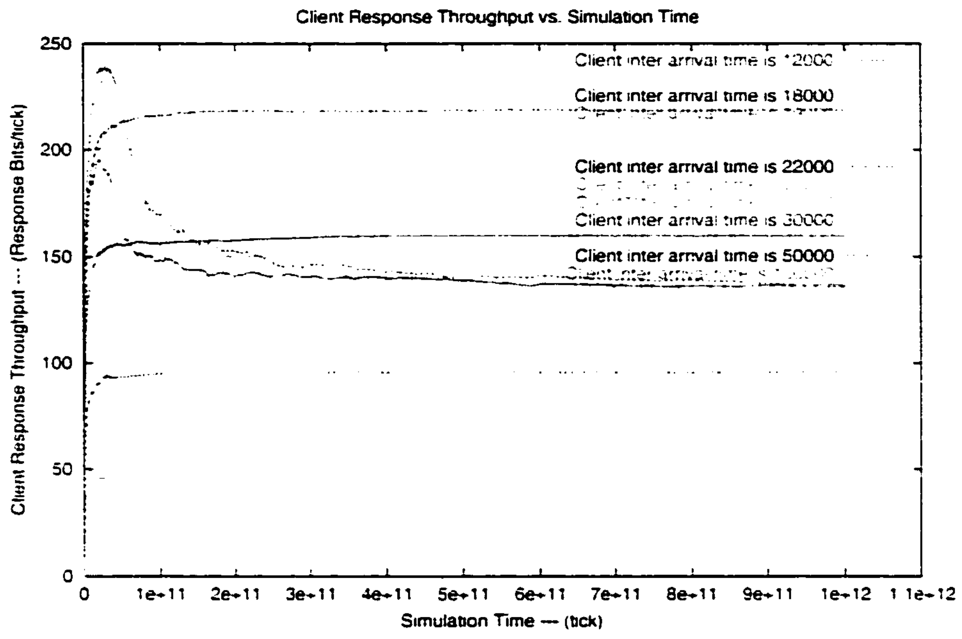


Figure 4.1: Throughput against Time T_i

interval represents an interval that a specified random variable will fall within with a given level of confidence. For example, suppose X is a random variable. Then a 95% confidence interval would consist of an interval such that there is a 95% probability that X will fall within that interval. We take the following steps to get the confidence interval.

1. First for any result x_i $(1 \leq i \leq m)$ we produce, we need to compute a mean for all replications:

$$\bar{x} = \frac{1}{m} \sum_{1 \leq i \leq m} x_i \tag{4.3}$$

where m is equal to 8 in our simulation.

2. Calculate the variance for the mean of the replications:

$$Var(\bar{x}) = \frac{1}{m-1} \sum_{1 \leq i \leq m} (x_i - \bar{x})^2 \tag{4.4}$$

where m is equal to 8.

3. Then the $100(1-\alpha)\%$ confidence interval for the mean value is given by

$$\bar{x} \pm t_{[1-\alpha/2;n-1]} \sqrt{Var(\bar{x})} \tag{4.5}$$

where $t_{[1-\alpha/2; n-1]}$ is the $1-\alpha/2$th quantile of a t distribution variate. We compute the 95% confidence interval for each mean value. Thus α is equal to 0.05 and $z_{1-\alpha/2}$ is the 0.975th quantile of a t distribution variate, which equals 2.306.

4.3 Simulation Implementation

In order to evaluate our proposal we first studied the response of a traditional network, which was subjected to the network load generated by crawlers. We then compared its behavior to the one of our proposed active network with packet monitoring and page indexing on active gateway routers.

As described in Section 3.2, the crawler request hit percentage averages 27.3% and reaches 40.6% on the Web server of our department. So for a traditional network, we simulated systems with 0% crawler traffic, 20% crawler traffic and 40% crawler traffic. By 40% crawler traffic, we mean the load caused by crawlers is 40% of the total network load in terms of the request arrival rate. Our simulation is based on the assumption that the network load is only comprised of the crawler load and the client load which is generated by a variety of Web clients from all over the world. So on a network with 40% crawler traffic, crawler load is around 40%/(100%-40%)=66.7% of the client load. By comparison among 0%, 20% and 40% cases, we will see how much the system behavior is affected when exposed to different rates of crawler traffic.

We assume that there is only one gateway for each stubnetwork. The gateway sees all the traffic between its stubnetwork and the rest of the world. However in reality, there should be more than one gateway for each stubnetwork, each gateway monitors the traffic between the stubnetwork and part of the world. Therefore, it is possible for a Web page to be indexed on the multiple gateways of the same stubnetwork and the removal of duplicated pages must involve the cooperation among the gateways. In this way, the indexing data delivered to the repository would be still at most 7% of the original HTML files and the network bandwidth is saved.

In Section 3.3.5, we showed that if active routers index all passing by HTTP traffic. then the compressed inverted file that is sent to the repository is only about 7% of the size of the original HTML documents. For all the passing HTTP messages. we only need to index those pages which have been modified since they were last indexed and new pages. Therefore. the extra indexing data sent to the repository should be considerably less than 7% of the size of the original HTTP messages. Thus. for the case of the active network. we simulated systems with 0% overhead. 3% overhead and 7% overhead. By overhead. we mean the bytes generated by indexing as a fraction of the number of bytes in the passing Web pages. Note that the 0%/3%/7% values in the active indexing case represents the efficiency of indexing. whereas the 0%/20%/40% values in the with-crawler case indicate traffic intensity. In the cases of the 0% active indexing and the 0% with-crawler. the workloads generated by all human clients are identical and no other traffic is imposed on the network. Therefore. although we implement these two cases in different simulations. they should have the same performance. For other cases. active indexing cases should have overall better performance than those of the with-crawler cases. We will see in Section 4 of this chapter how much the performance is affected by comparing all the active indexing cases with all the with-crawler cases.

4.3.1 Network topology

We chose the transit-stub model generated using the Georgia Tech Internet Topology Generator(GT ITM) [24] to create a network that resembles the structure of the Internet. The transit-stub model is composed of interconnected transit and stub domains. A transit domain comprises a set of highly connected backbone nodes. represented by squares in Figure 4.2. A backbone node is either connected to some stub domains or other transit domains. A stub domain usually has one or more gateway nodes. which have links to transit domains. The Internet is regarded as a collection of interconnected routing domains. which are groups of nodes that are under the same administration and share routing information. The topology generator follows the following routing rules:

- The path connecting two nodes in a domain stays entirely within the domain.

- The shortest path between node u in stub domain U and node v in stub domain V goes from U through one or more transit domains to V, and does not go through any other stub domains.

- If two stub domains are connected directly by a stub-stub edge, the path between nodes in the two domains may go through that edge and does not pass through any transit domains.

The following steps are taken to construct the model. First, the topology generator construct a connected random graph where each node represents a transit domain. It then replaces each node in the graph with another connected graph to represent a transit domain. For each node in the transit domains generate a number of connected random graphs to represent stub domains attached the transit node. Finally additional edges are added to connect transit nodes to their corresponding stub domains and for some stub domains, to connect two stub domains together.

Once the graph is completed, integer edge weights are assigned in such a way that the path between any two nodes in the same domain will remain within the domain.

As shown in Figure 4.2, we use a graph which consists of one transit domain of 4 nodes. Each node in the transit domain has a number of stub networks connected to it. There are no edges between any two of the stub domains. The topology contains 100 nodes with an overall average node degree of 3.74.

4.3.2 Network configuration

Both the traditional and active networks utilize the same topology, which is depicted in Figure 4.2. On the former type of network, we define five kinds of nodes: core routers, gateways, crawlers, servers and clients. The four transit nodes represent core routers. Each stub domain contains a gateway to the

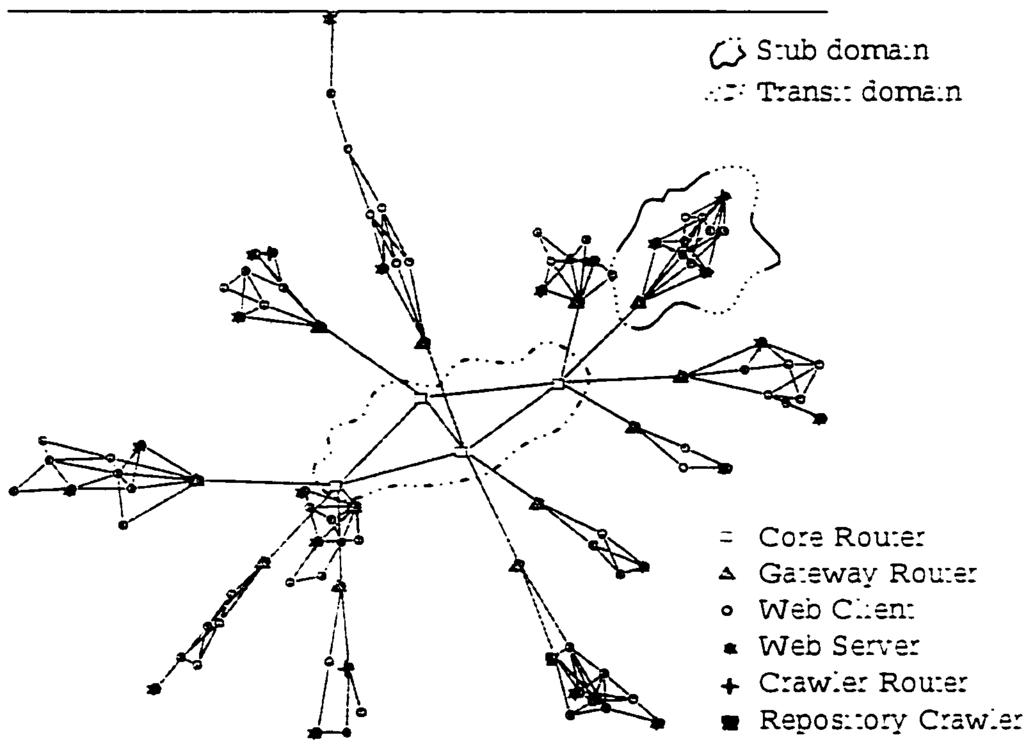


Figure 4.2: Transit-Stub Study Network

transit domain. Each gateway node forwards packets out of and into its stub domain. The node allocation for the traditional network is illustrated in Figure 4.2. We have 12 gateway nodes since there are 12 stub domains in the topology. Four crawlers are located in the four different stub domains. Finally 20 servers and 60 clients are distributed among stub domains.

For the active networks case, the four crawlers are replaced by three regular routers and one backend repository. Regular routers are different than core routers in the sense that the link speed between core routers is much faster than the one between regular routers and their adjacent nodes. The difference between regular routers and clients is that regular routers only forward packets whereas clients initiate HTTP requests and route packets as well. The backend repository receives and stores indexing data sent by gateways. Gateways on the active network not only forward packets, but also generate extra indexing data periodically, which is bound for the backend repository. The node allocation for the active network is also shown in Figure 4.2

4.3.3 Simulation Parameters

Session and Client Request Arrivals

The series of requests made by a single user to a single site is viewed as a logical session. In the course of a session, the user may perform one or more clicks to ask for Web pages. A click triggers the browser to issue a HTTP request for a resource and subsequently more requests if there are any embedded resources referenced in that page. Studies have shown that the session interarrival times follow an exponential distribution[26]. Exponential interarrival times correspond to a *Poisson process*, where users arrive independently of one another. In our simulation model, we made the assumption that there is only one request issued throughout a session. Therefore, HTTP request arrivals are also modeled after an exponential distribution. We vary the mean client request interarrival time to impose different workloads on the network.

The total client requests are the total client requests generated by all 60 clients until the time when the simulation terminates. Let total client request arrival rate be *clientArrivalRate* and total client requests be *totalClientRequests*. The *clientArrivalRate* is given by:

$$clientArrivalRate = \frac{totalClientRequests}{totalSimulationTime} \tag{4.6}$$

Request Sizes

Each HTTP request packet size is approximately lognormal distributed with mean=2880 bits and standard deviation=848 bits[27]. The simplest way to generate a lognormal distribution is to use a normal one. If Y is normally distributed. then X = exp(Y) is lognormally distributed. Since [27] gives the mean and standard deviation of the lognormal distribution. if we want to generate the lognormal through a normal distribution. we need to figure out the mean and standard deviation of the corresponding normal distribution. Since we know if Y is normally distributed with mean μ and standard deviation d. then the mean μ_L and standard deviation d_L of X is expressed as:

$$\mu_L = e^{\mu-\frac{d^2}{2}} \tag{4.7}$$

$$d_L{}^2 = e^{2(\mu-d^2)} - e^{2\mu-d^2} \tag{4.8}$$

$$\begin{aligned} \mu_L &= e^{\mu-\frac{d^2}{2}} \\ ------> \mu = \ln(\mu_L) - \frac{d^2}{2} \\ d_L{}^2 &= e^{2(\mu-d^2)} - \mu_L{}^2 \end{aligned} \tag{4.9}$$

therefore. we have

$$\begin{aligned} \ln(\mu_L) &= \mu + \frac{d^2}{2} \\ \frac{\ln(d_L{}^2 + \mu_L{}^2)}{2} &= \mu + d^2 \end{aligned}$$

use (4.9) to replace μ

$$\begin{aligned} d^2 &= \ln(\mu_L{}^2 + d_L{}^2) - 2\ln(\mu_L) \\ &= \ln(\frac{d_L{}^2}{\mu_L{}^2} + 1) \end{aligned}$$

finally, we have

$$\begin{aligned} d &= \sqrt{\ln(\frac{{d_L}^2}{{\mu_L}^2} + 1)} && (4.10) \\ \mu &= \ln(\mu_L) - \frac{d^2}{2} \\ &= \ln(\mu_L) - \frac{\ln(\frac{{d_L}^2}{{\mu_L}^2})}{2} && (4.11) \end{aligned}$$

Crawler Request Arrivals and Total Crawler Request Arrival Rate

Crawler request interarrival times also follow an exponential distribution since crawlers are just more heavy demand clients. Let *crawlerMIT* be the mean crawler request interarrival time, *clientMIT* be mean client request interarrival time and *crawlerRate* be crawler traffic rate on the network. There are 4 crawlers and 60 clients in the traditional network. The *crawlerMIT* is given by the following formula:

$$crawlerRate = \frac{\frac{4}{crawlerMIT}}{\frac{4}{crawlerMIT} + \frac{60}{clientMIT}} \qquad (4.12)$$

$$--> crawlerMIT = \frac{clientMIT * (1 - crawlerRate)}{15 * crawlerRate} \qquad (4.13)$$

The total crawler requests are the total requests generated by all 4 crawlers until the end of the simulation. Similarly, each crawler generates requests following an exponential distribution with mean crawler request interarrival time, *crawlerMIT*. Let total crawler request arrival rate be *crawlerArrivalRate* and total crawler requests be *totalCrawlerRequests*, then *crawlerArrivalRate* is given by:

$$crawlerArrivalRate = \frac{totalCrawlerRequests}{totalSimulationTime} \qquad (4.14)$$

Response Sizes

The response sizes can be represented as a combination of lognormal distribution and a heavy-tailed distribution[15]. The body of the distribution is represented by a lognormal distribution and the tail fits a heavy-tailed distribution. The heavy tail refers to how slowly $F(x)$ decrease for larger values of x. However, in our simulation, we only represent the response sizes as a lognormal distribution with Mean = 80,000 bits and standard deviation(SD) = 200,000 bits[27]. Given the Mean and SD, we can use the same method used to compute request sizes to calculate the response size for each response a server generates. The great variability in sizes of different types of content results in the large standard deviation for response sizes.

Indexing Message Sizes and Inter-departure Times

The indexing message length is fixed with the length=10,000bits. The message inter-departure time is unique for each gateway on a specific simulation case. Since we send the indexing traffic to the backend repository constantly with the dribble mode, theoretically gateways should spend the same time to transmit the indexing data as what was used to accumulate them. Although each gateway generates different amount of indexing traffic within a given time period, we make sure they are sent in the same fixed time. We balance the index accumulating time and the index delivery time so that there is no extra index data left on gateways. We could also vary the index transmission time according to the current network condition. We will address this in the future work of Chapter 5. So each gateway should send the indexing data with its own inter-departure rate which is proportional to the volume of its accumulated indexing traffic. In our simulation, at time *4e+11* all gateways begin to gather the passing HTTP messages to produce indexing information until time *5e+11*, from which point to time *6e+11*, they deliver the indexing data to the backend repository.

Link Speeds

Due to the characteristics of the Internet, backbone routers are usually high-speed devices while most of the stub nodes are regular hosts. We set different link speeds for different types of links. The link speed is represented as transmission rates of the two ports attached to the link. The port transmission rates for a link are the same. The transmission rate of the ports attached to the core links is set to 100Mbps. the transmission rate of the ports attached to the links between gateways and core routers is defined as 10Mbps. and we set the transmission rate of the ports attached to the stub links to 1Mbps.

4.3.4 Performance Metrics

Throughput

We distinguish throughput caused by normal clients and throughput resulting from crawlers. Client throughput is the number of bits of responses received by normal clients per simulation time. Similarly. Crawler throughput is the number of bits of responses arriving at crawlers per simulation time.

- **Client throughput** In active indexing. the throughput we computed is called *active throughput*. In the cases of 3% and 7% overhead. we calculate the average throughput over a period from the point when gateways start sending the indexing traffic until all the indexing traffic has been received in the repository. For all cases of different request arrival rates. gateways start transmitting indexing traffic at the same time. at time *5e+11*. and stop the transmission at *6e+11*. In the case of 0% overhead. since no extra traffic is generated and the simulation has long reached the steady state at time *5e+11*(in Figure 4.1). we measure the average throughput from the time *5e+11* until the simulate terminates at time *10e+11*. In the with-crawlers cases. in order to compare different performance. we calculate the average client throughput between time *5e+11* and time *10e+11* when the simulation ends. The different time periods over which the client throughput is computed for the simulation cases are illustrated in Figure 4.3.

  Although in all 3% and 7% active indexing cases. the gateways spend the same amount of time to send the indexing data. the time when all the indexing traffic have been received should vary according to request interarrival rate. The simulation times against different percentages of received index for 3% and 7% cases are illustrated in Figure 4.4 and

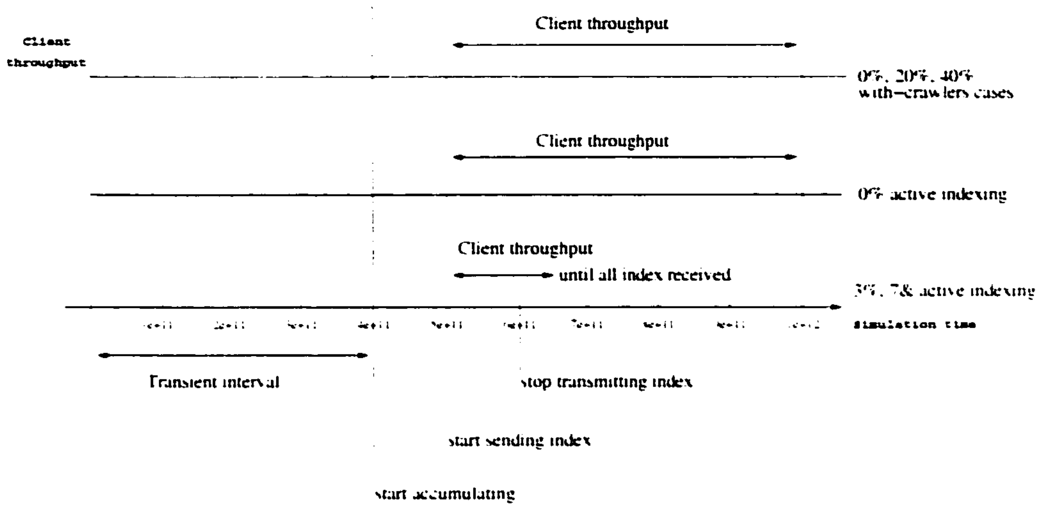


Figure 4.3: Client throughput vs. timeline

Figure 4.5 respectively. From the two figures, we can see that in the cases with relatively small request arrival rate, the indexing traffic are all received shortly after gateways stop the transmission at *6e+11* and the index receiving times are linear to the percentage of received index. However, after the request arrival rate reaches some threshold, the index receiving times increase exponentially with only a portion of total index data received when the simulation terminates. In these cases, it is hard to estimate when all the index data will be received.

In active indexing, we accumulate the indexing data over a long time period, and deliver the data to the repository in the next same amount of time period. We run 8 independent runs for each of our calculated data points. For each experiment, we observed the commonly accepted simulation methodology [4] ensuring proper transient removal and termination of our simulation. Due to the limited computational resources at our disposal, we opted for repeating our sets of experiments several times instead of conducting fewer runs for a very long period of time. Both methods are considered equivalent in the [4] as far as their accuracy is concerned. Also, our measured 95% confidence intervals were less

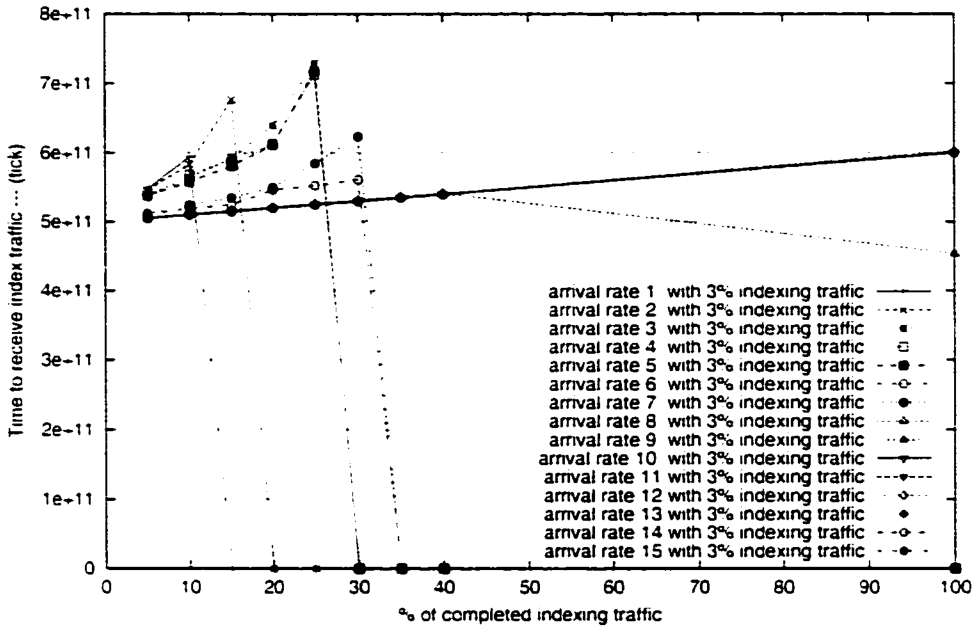


Figure 4.4: different index receiving times for all 3% active indexing cases

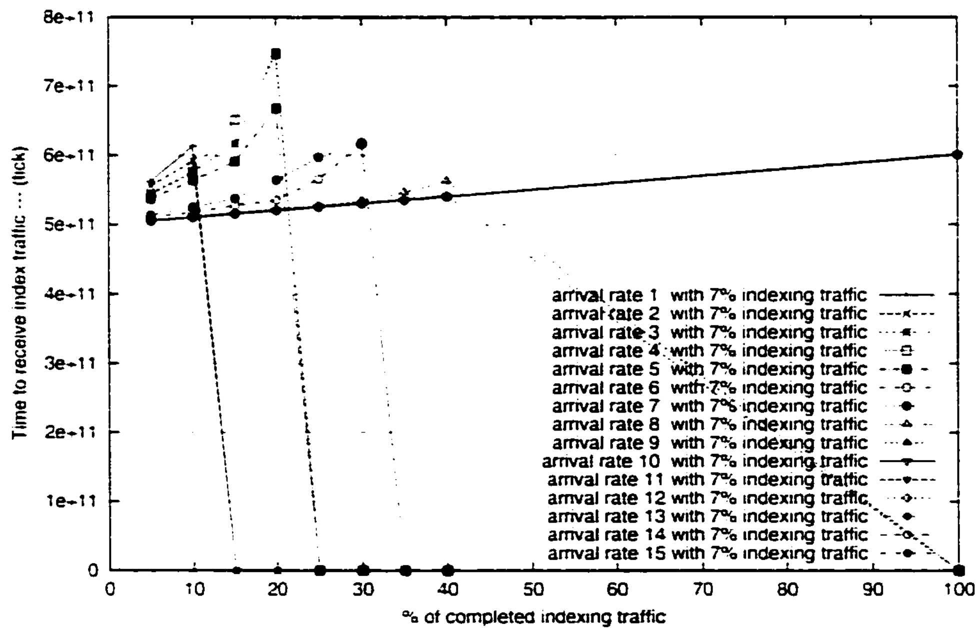


Figure 4.5: different index receiving times for all 7% active indexing cases

than 5% indicating that our estimated means are accurate. The use of several runs lessens the dependence of our results on the initial seeds for the pseudo-random number generator. Furthermore, even in the 0% active indexing case with the smallest request arrival rate, the number of total requests reaches 30210. Therefore, there are about 3000 requests generated from time *5e+11* to time *6e+11*, one tenth of the total simulation time when the index data are gathered. Therefore, the volume of the accumulated index data is not trivial. From all the arguments given above, several independent repeated runs are at least as good as, if not better than, one or more longer runs.

- **Crawler throughput** In the with-crawlers cases, we also compute the throughput of responses generated by crawlers. The average crawler throughput is attained in the period from time *5e+11* to time *10e+11*.

Average Request Delay

We are more interested in what clients see after they issued a request. The *Average Request Delay* is the average number of simulation time units from the point of a request's generation until its corresponding response has been received. We compute average client request delay, in the with-crawlers cases as well as average crawler request delay over the period between time *5e+11* and time *10e+11*.

Request Completion Rate

Request completion rate is the total completed requests divided by total generated requests. The complete rate is equal to 1 if all generated requests have been served and the responses have been delivered back to clients.

4.3.5 Routing in the simulation

When a router receives a new packet, it always dynamically calculates the total lengths of possible paths to find an idle output port with the minimum value of the cost rank in terms of the link length for a given destination. If

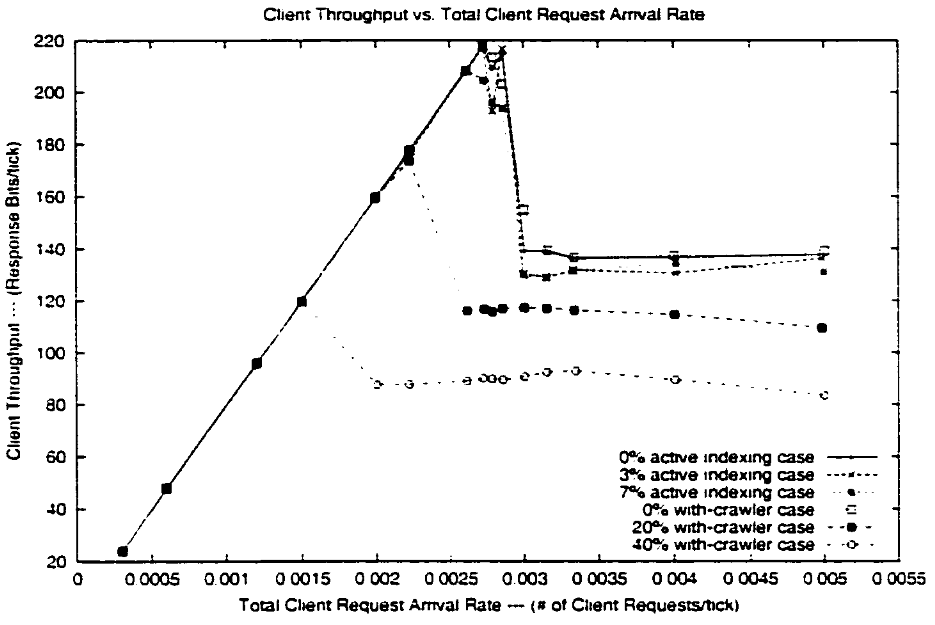


Figure 4.6: Client Throughput in All Cases

there is more than one port with the same least cost. one of them must be chosen at random so that no port is privileged or discriminated against. If all the output ports are occupied. then the packet will be queued until an output port is available.

4.4 Experimental results and analysis

As described before. we simulated the active network with 0%. 3% and 7% extra indexing data respectively and the traditional network with 0%. 20% and 40% extra traffic initiated by crawlers. At 3% indexing data. we should see the routers generating (and the repository receiving) 3% of the byte rate received by the clients in the with-crawler case at 0% crawler activity. At 7%. we should see the routers generating. and the repository receiving. 7% of the byte rate received by the clients in the with-crawler case at 0% crawler activity. We will compare the network performance in terms of the number of bits per unit time that are received by clients. how quickly a request is met (average request delay) and the completed client request(the ratio of completed requests to all the requests).

Figure 4.6 illustrates the different client throughputs on both the traditional network and the active network. Figure 4.7 only makes a comparison about throughput among active network cases while Figure 4.8 shows different throughputs on the traditional network. In these figures. the horizontal axis represents the total client request arrival rate. i.e.. the number of total client requests generated on the network per simulation time unit. and the vertical axis shows the throughput - the number of bits of responses received by all the clients per simulation time unit. From Figure 4.6. we can see the throughput for the 0% active indexing matches the one for 0% with-crawler case. This verifies that the simulations for 0% active indexing case and for 0% with-crawler case should have the same performance. Both of them achieve the same peak throughput at about 222 bits/tick. after which point. the throughput drops down quickly when the systems becomes saturated. After the systems get overloaded. the throughput remains steady at about 140 bits/tick whatever the client request arrival rate is.

We define the maximum supported client arrival rate as the client request arrival rate at the point where the client throughput reaches the maximum value. In Figure 4.7. compared with the 0% active indexing case. the 3% active indexing case reaches its threshold slightly earlier. which means the maximum client request arrival rate the active network with 3% indexing traffic can support is slightly smaller than that of the active network with 0% indexing traffic. Thus. we show that the client throughput for the 0% active indexing case is slightly larger than the 3% active indexing case. With indexing traffic rate increasing to 7%. the maximum supported client arrival rate decreases.

Also in Figure 4.6. compared with the active network. the traditional networks with crawler rate 20% and 40% have much less maximum supported client arrival rate. The reason is that although we add some indexing traffic to the network after the crawler traffic is removed. the overall network traffic is still much less than that of the traditional network since the traffic introduced by the crawler constitute a large part of the total network load. Also. we can find that as the crawler rate increases. the maximum supported client arrival rate decreases.

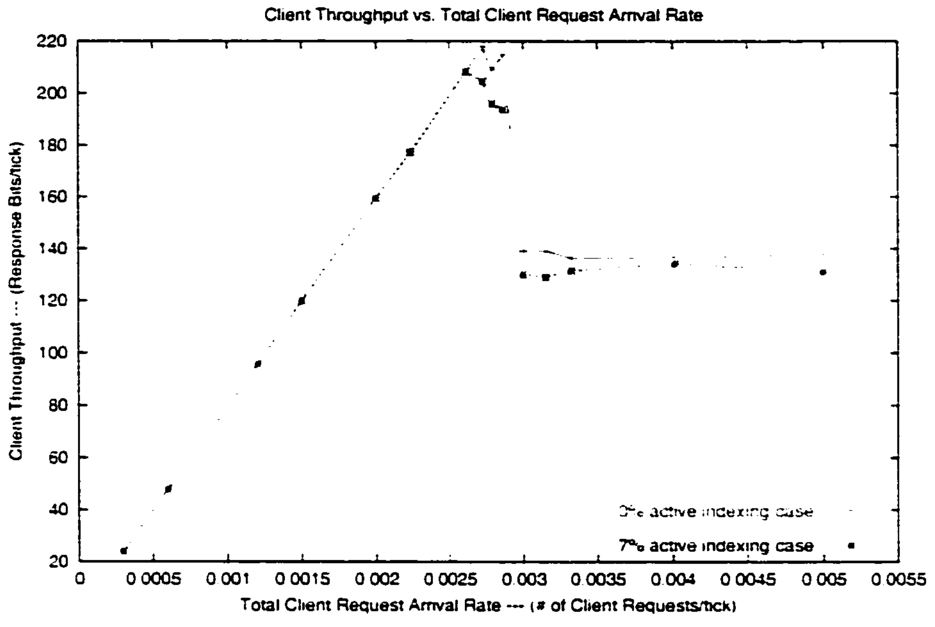


Figure 4.7: Client Throughput in Active Indexing cases

It is also shown in Figure 4.6 that 3% and 7% active indexing cases have a maximum client throughput that is a little bit smaller than that of the 0% active indexing while the 40% with-crawler case attains the least maximum client throughput among all the cases.

Figure 4.9 shows the crawler throughput in the 20% and 40% with-crawler cases. The vertical axis represents the number of bits of responses per simulation time unit received by crawlers and the horizontal axis shows the total request arrival rate which is the number of requests generated per unit time by both human visitors and crawlers. It is illustrated in the graph that crawler throughput in the 40% case is overall twice that in the 20% case. In the 40% case. the crawler load is 66.7% of the traffic generated by a variety of human visitors while in the 20% case. the crawler load weighs 25% of the client traffic. So the crawler load in 40% is 66.7%/25.%=2.64 times the crawler load in the 20% case. However. the 40% case achieves overall worse performance. in our case worse throughput. than the 20% case. That explains why the crawler throughput of the 40% case is only about twice rather than 2.64 times that in the 20% case. Also in the Figure 4.9. the 40% case reaches the maximum

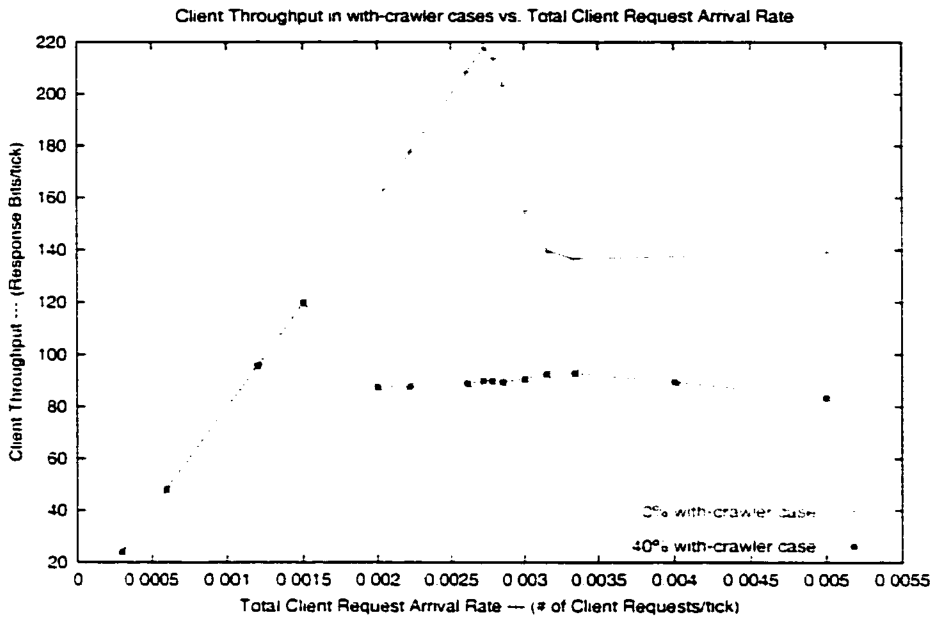


Figure 4.8: Client Throughput in With-crawler Cases

crawler throughput slightly earlier than the 20% case. This is also because the 40% case is more overloaded. thus the network gets saturated earlier.

Figure 4.10 demonstrates the average client request delay for all the with-crawler and without-crawler cases. Figure 4.11 and Figure 4.12 show the average client request delay corresponding to with-crawler cases and active indexing cases respectively. The vertical axis shows the average number of simulation time units for a client to wait to receive a response after it sends a request. The horizontal axis is the number of total requests per time unit generated by human visitors. In Figure 4.10. we can find that the curve corresponding to the 0% active indexing case is coincident with the one associated with the 0% with-crawler case.

It is shown from Figure 4.10 that for a low-loaded network (when the client arrival rate is small). the average client request delays for the 3 active indexing cases are almost the same at the same client request arrival rate. However. the traditional network with 20% or 40% crawler traffic still has slightly longer average delay than that of the active network. That tells us that the traditional network has more overall network traffic for a certain client arrival rate than

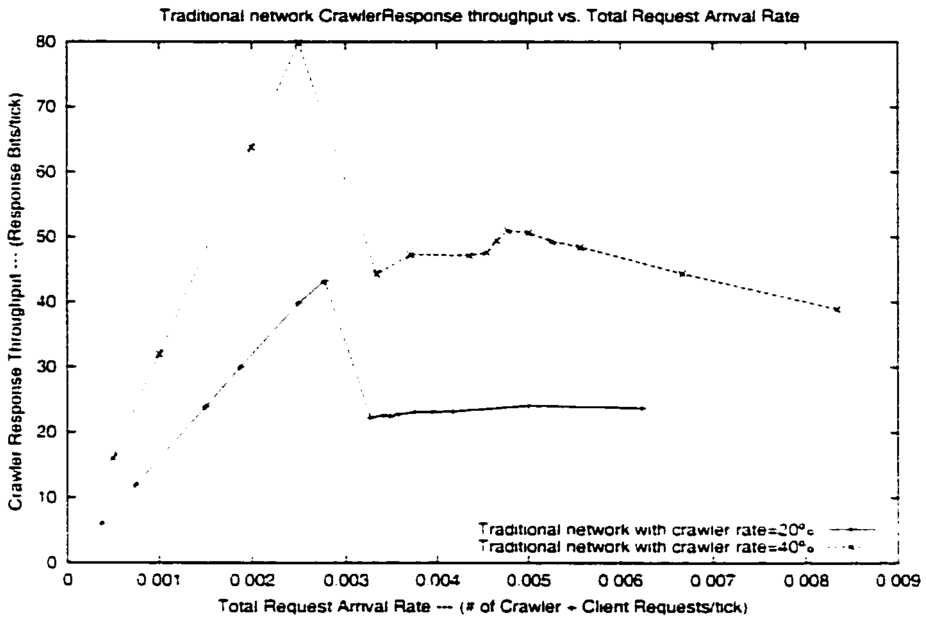


Figure 4.9: Crawler Throughput in With-crawler Cases

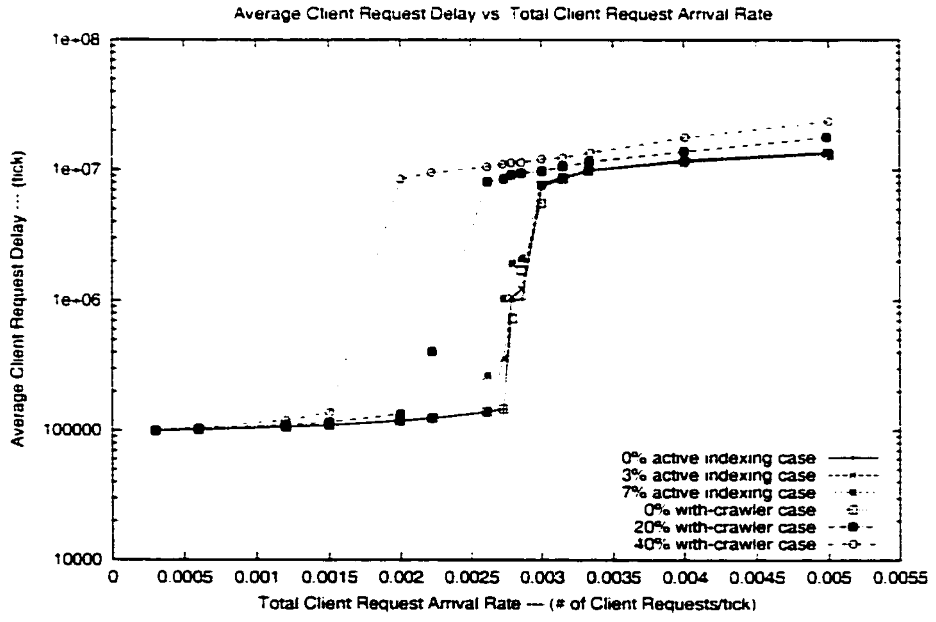


Figure 4.10: Average Client Request Delay in All Cases

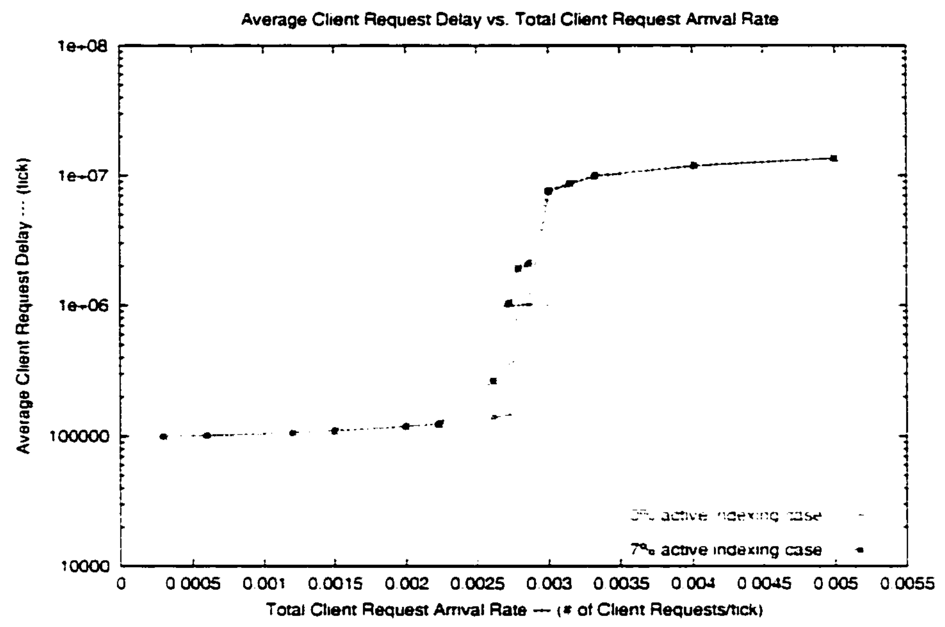


Figure 4.11: Average Client Request Delay in Active Indexing Cases

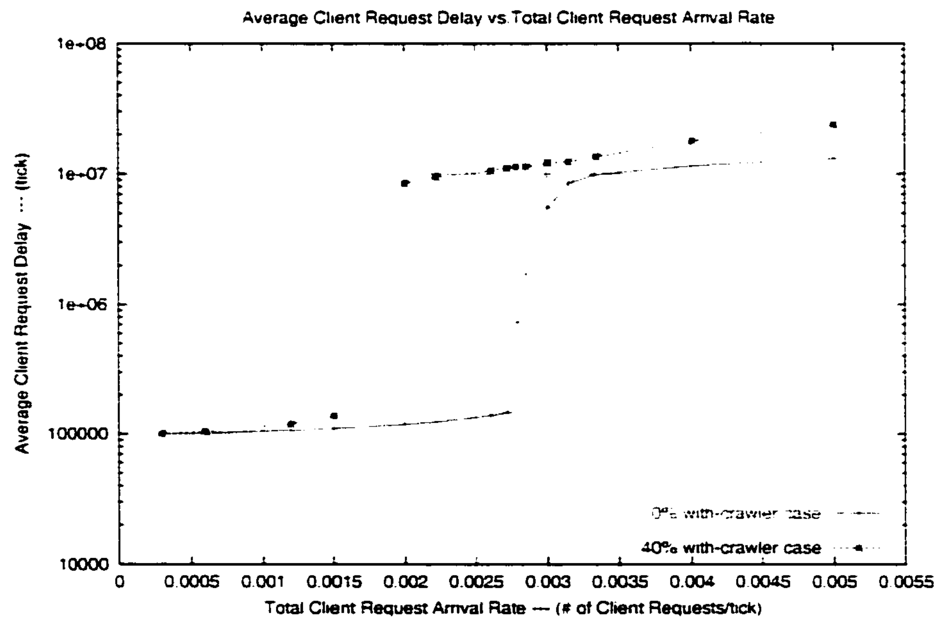


Figure 4.12: Average Client Request Delay in With-crawler Cases

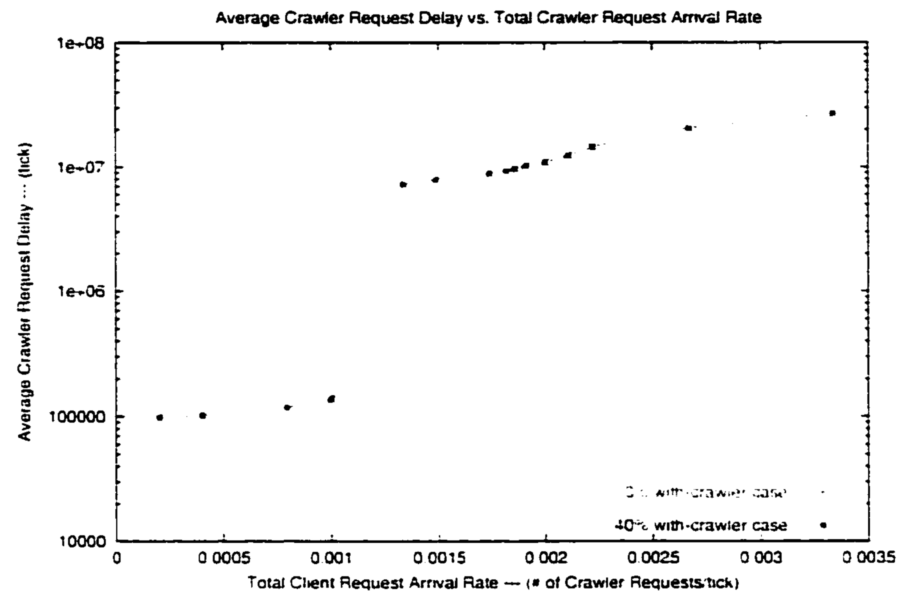


Figure 4.13: Average Crawler Request Delay vs. Total Crawler Request Arrival Rate in With-crawler Cases

the active network. On the low-loaded network, since the system has enough bandwidth, the indexing traffic will hardly affect the performance since it is ignorable compared with the bandwidth. Since there is relatively more traffic introduced by crawlers, the average delays for the traditional network are slightly longer. Only at a very low client arrival rate(the first point in all the curves), do the six cases have the same average delay.

With the client request arrival rate increasing, the average delay begins to soar at different client request arrival rates with the 40% with-crawler case the least and both the 0% with-crawler case and the 0% active indexing case the most. After the network load becomes saturated, the 40% with-crawler case has the longest average delay and the 0% with-crawler case and the 0% active indexing case have the shortest average delay.

Figure 4.13 illustrates the average crawler request delay, i.e. the average number of simulation times for a crawler to wait to receive a response after sending a request. The x axis represents the total crawler request arrival rate. The graph may gives us a first impression that the 20% with-crawler case has a higher average request delay than the 40% with-crawler case for the

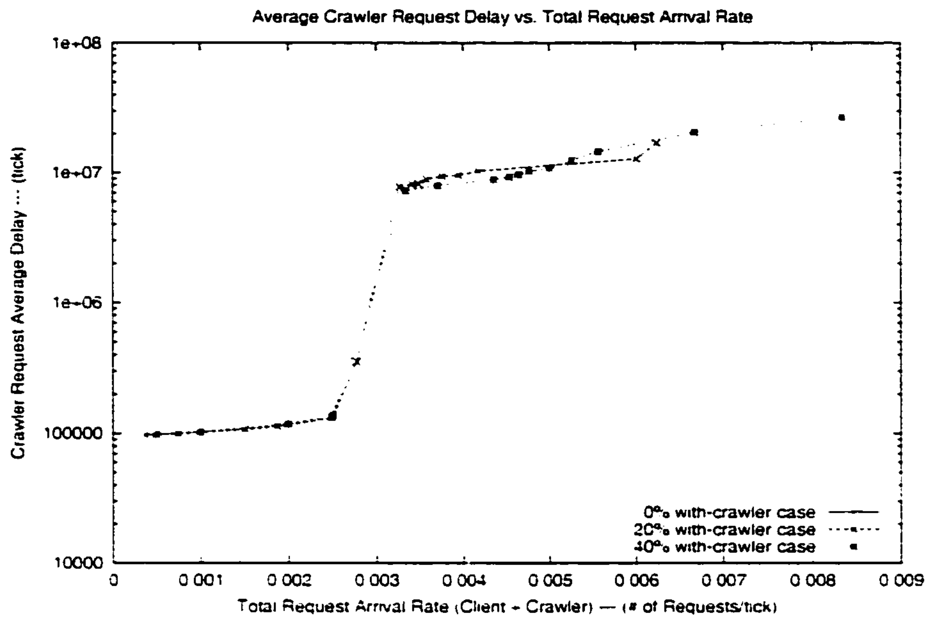


Figure 4.14: Average Crawler Request Delay vs. Total Request Arrival Rate

same crawler request arrival rate. However if there are 10 crawler requests generated in a given time period. the number of total requests initiated is 50 in the 20% with-crawler system and 25 in the 40% with-crawler system. At the same crawler request rate. the 20% with-crawler system should be much more loaded than the 40% with-crawler system. That explains why the curve for the 20% case seems steeper than the curve for the 40% case. We base our simulation on the same total client request rate. So the largest crawler arrival rate in the 40% case is 0.0033 which is around 2.64 times 0.00125 the largest crawler arrival rate in the 20% case.

Figure 4.14 shows the average crawler request delay versus the total requests arrival rate. From the graph. we can see that the two curves are nearly identical. This is because we are comparing the system performance when the total load for the 20% case is identical to the one for the 40% case. When the total loads for the two systems are the same. the performance. crawler request average delay in our case. should be the same. Then 20% of the total load should have the same performance as 40% of the total load. although the crawler request rates are different in the two cases.

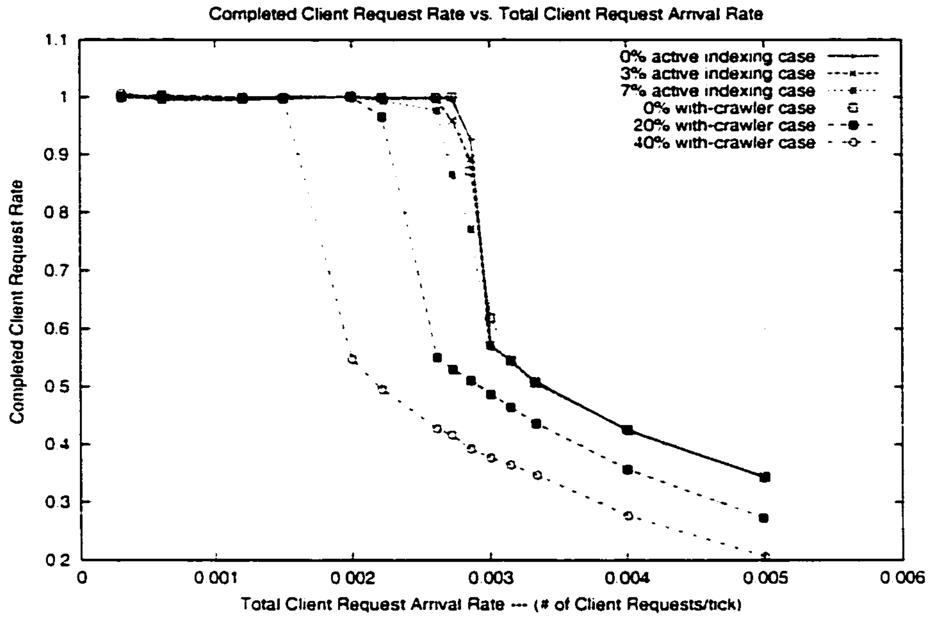


Figure 4.15: Completed Client Request Rate in All Cases

Figure 4.15 illustrates the ratio of completed client requests to all the client requests. From the graph. we can see that almost all the requests are satisfied when the request arrival rate is small. When the systems get overloaded at some point. the completed request rates begin to decrease. The rate starts to drop down at different client request rates for different systems and it declines earliest in the 40% with-crawler system and latest in both the 0% with-crawler and 0% active indexing systems.

4.5 Summary

We simulated two different systems: the traditional network with crawlers and the active network without crawlers. but with indexing data sent from gateways to the backend repository. From the simulation results. we achieved significant gains in terms of the user perceived latency and the system throughput caused by human visitors. Even in its worst case when the volume of indexing information is 7% size of original HTML documents. the effect of the indexing data sent to the repository is trivial compared with all the with-crawler cases. The removal of the crawler traffic enables Web servers to deal with human requests more efficiently. It relieves the underlying network of the extra traffic from crawlers and the rest of the data on the network can be transmitted faster.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis. we propose a more efficient indexing system than what is currently being used by search engines. Search engines rely on crawlers to collect Web pages for their database. whereas our system uses packet monitoring on strategic nodes on the Internet to gather information and eliminate crawler traffic. Our approach gains significant advantages over the current one deployed by search engines. Firstly. by removing the crawler traffic. the network bandwidth is saved and made available to human requests. Given the limited bandwidth of the Internet. fewer packets will be dropped because of congestion. The network will be more efficiently utilized and move the rest of the data faster from sources to destinations. Furthermore. servers' resources such as CPU cycles are saved so that the servers can be dedicated to processing a variety of human requests more quickly. which leads to the reduction of user perceived latency and improves the throughput of responses introduced by human visitors.

Table 5.1 illustrates the comparison among the client throughput in the 40% with-crawler case. the 7% active indexing case. and 0% active indexing case with various client request rates. From the experiments on the logfiles of our department Web server. the crawler percentage hits has a maximum of 40.6%. so we use the 40% with-crawler case for the comparison. Table 5.1 gives us the absolute impact of both crawlers and the active indexing. A few observations can be made from the table. Firstly. the maximum throughput

Client Request Rate	T in 40% with-crawler	T in 7% active	T in 0% active
0.005000	83.417839	131.05	137.655548
0.004006	89.440071	134.20	136.615479
0.003343	92.806702	131.60	136.252182
0.003150	92.456474	129.05	139.020096
0.003002	90.604248	129.90	139.023026
0.002858	89.466873	193.75	214.858551
0.002785	89.761986	196.05	208.978912
0.002721	89.966042	204.5	217.786285
0.002612	88.931145	208.4	208.285995
0.002220	87.637299	177.190292	177.410156
0.002000	87.518570	159.425034	159.633148
0.001502	119.749161	119.910873	119.799004
0.001200	95.726089	95.754204	95.773140
0.000601	47.996319	47.847134	47.957947
0.000298	23.989302	23.859081	23.906977

Table 5.1: Comparison among client throughput in 40% with-crawler. 7% active indexing and 0% active indexing case(T represents client throughput)

in the with-crawler cases. 119 bits/tick. is much smaller than the one in the 0% active indexing cases. 217 bits/tick. The table also shows that. when the client request rate is larger than 0.001502. all the client throughputs in the with-crawler cases are considerably less than the ones in the active indexing cases. When the network is lightly loaded. the extra load caused by crawlers does not have much impact on the network performance. in this case the speed at which clients receive their responses back. However. with the client request rate increasing. the network gets more congested with more responses generated by Web servers to be transported. The situation is deteriorated by the excess traffic resulted from crawlers. Now let us look at the difference between the throughput in the 7% and the 0% active indexing cases. Table 5.1 shows that the maximum throughput in 7% the active indexing cases. 208 bits/tick. is slightly smaller than the one in the 0% active indexing cases. 217 bits/tick. When the client request rate is larger than the threshold - 0.002612 - at which point the throughput reaches the maximum in the 7% active indexing cases. the throughput in the 7% active indexing cases is a little bit smaller than the one in the 0% active indexing cases. Therefore. although a small amount of additional index data is levied on the Internet. it does not influence the network performance much compared to the crawler traffic. With the removal of crawler traffic from the Internet. it is capable of moving the Web traffic faster caused by various human users and the user's requests are. hence. served more quickly.

In addition. due to the strategic location of gateway routers watching out for passing traffic. our system can index Web pages more exhaustively than crawlers do. Crawlers follow links on the Web to find new pages. As explained in Section 3.3.1. the chance of a page being fetched crawlers is contingent on its link relation to other pages and the link structure of the Web. Therefore. some pages are never captured. In our scheme. each gateway takes charge of pages on the servers in its adjacent stub networks. Therefore. each page on the Internet has an opportunity to be retrieved regardless of its link relation to other pages.

Apart from the existing statistical data used in responding to a user query by search engines. our approach also can fetch dynamic data about a page. such as the number of visiting times globally per unit time. The dynamic data about a page. can be used in conjunction with the existing statistics to rank the query results. In this way. the results ranked on the top tend to be of most interest to users.

Finally. although our indexing system obviates the need of crawlers. hence removes the crawler traffic from the Internet. the system is not able to capture two types of Web resources. One type is new pages which have not been visited by human clients. The other one is those pages which have been indexed before. but have been modified and have not been requested by clients since the modification.

5.2 Future Work

There are several issues we want to address about future work. We use dribble mode to deliver the indexing data from gateways to the backend repository. The indexing data is transmitted at a rate so that gateways use the same amount time to send the indexing traffic as what has elapsed to accumulate it. To put it briefly, gateways constantly forward indexing information to the repository. When the network is lightly loaded, the constant extra traffic does not pose a significant load to the network. However, if the network is already overloaded, the extra traffic may deteriorate the situation. In real life, the traffic load on the Internet varies from time to time. One possible future research venue is to adjust the transmission rate according to the current condition of the network. During the periods of low network load, gateways could send the indexing data at a higher rate. If the traffic gets heavier, gateways would not send data or send data at a lower rate until the network becomes uncongested again. Under different network conditions, it might be possible for gateways to optimize the transmission rate of indexing information both in theory and in practice. Factors taken into account include current network workload, desired updating rate of the database, the volume of the indexing data to be delivered and how much the network performance would be degenerated. Therefore, we could make a good trade-off between the deterioration of network performance as a result of immediate transmission of indexing data by gateways and the freshness of our collection about Web resources.

In order to see how much achievement has been gained by using our active indexing system, our experiments were conducted on a simulation tool - SMURPH rather than on a real-world network. It would be interesting to do experiments on real networks. We could either set up a small scale network or use the current Internet backbone as our testbed.

In Chapter 3, we described a novel indexing system. But we only simulated the traffic volume in different cases and gave our analysis about network performance. The real implementation of the whole system was not done, such as the construction of HTTP messages from actual IP packets, building the inverted index from Web pages, gathering statistical data about Web pages, and so on. It would be desirable to implement and test the whole system under real-life conditions.

Our scheme is only able to keep up-to-date those pages which have been seen by gateways after its most recent modification. If some pages have been modified recently after they were last seen by gateways, the indexing system will not realize those pages in the database are already out-of-date. Also for those new pages which just appeared on servers and have not been requested by human visitors, gateways will not be able to catch these kinds of resources. To solve these problems, in addition to gateways passively watching out for passing traffic, each gateway could actively issue queries regularly to the servers in its stub-network and ask them to send new and recently modified pages to the gateway. Of course, this would involve closed cooperation between a gateway and its corresponding servers. In this way, we combine the passive monitoring of gateways and the active queries with servers to maintain our database as up-to-date as possible. We call this kind of scheme a hybrid approach.

Section 3.3.5 shows that if we build an index for all the passing pages whether they have been modified or not, the resultant indexing data is about 7% the size of original HTML documents. Usually a Web page should be visited more than once in its life span. Hence, if we only index a page once before it gets updated, the resulting index data should be less than 7% the size of Web resources in HTML. However, we do not know how much less than 7% of the original size it will be. Also from Section 3.3.5, we already know the average change frequency of a page in its category. If we can find out the average number of hits for a page in its life span and the distribution, we would be able to model our simulation more accurately and ascertain how exactly efficient our indexing scheme is.

Bibliography

[1] P. Gburzynski. "Protocol design for local and metropolitan area networks". Prentice Hall. 1996.

[2] J. Cho. H. Garcia-Molina. "Synchronizing a database to improve freshness". In Proceedings of the International Conference on Management of Data. 2000. http://www-diglib.stanford.edu/cgi-bin/get/SIDL-WP-1999-0116.

[3] J. Cho and H. Garcia-Molina. "The evolution of the web and implications for an incremental crawler". In Proc. of 26th Int. Conf. on Very Large Data Bases. pages 117-128. September 2000.

[4] R. Jain. "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design. Measurement. Simulation. and Modeling." Wiley-Interscience. New York. NY. April 1991.

[5] K. Claffy. G. Miller. and K. Thompson. The Nature of the Beast: Re cent Traffic Measurements from an Internet Backbone. In Proc. INET. July 1998.

[6] A. T. Campbell. H. G. De Meer. M. E. Kounavis. K. Miki. J. B. Vincente. and D. Villela. "A survey of programmable networks. " Computer Communication Review. vol. 29. no. 2. pp. 7-23. Apr. 1999.

[7] S. Bhattacharjee. Active Networks: Architectures. Composition. and Applications. Ph.D dissertation. http://www.cc.gatech.edu/projects/canes/bo bby/dissertation.html.

[8] K. Calvert. editor. Architecture Framework for Active Networks. h ttp://www.cc.gatech.edu/projects/canes/papers/arch-1-0.pdf.

[9] M. Hicks. P. Kakkar. J. T. Moore. C. A. Gunter. and S. Nettles. Ne twork Programming Using PLAN. http://www.cis.upenn.edu/ switchware/papers/progpl an.ps.

[10] M. Hicks. J. T. Moore. D. S. Alexander. C. A. Gunter. and S. M. N ettles. PLANet: An Active Internetwork. IEEE INFOCOM. New York. 1999. http://ww w.cis.upenn.edu/ switchware/papers/planet.ps.

[11] "Caml home page." pauillac.inria.fr/caml/index-eng.html.

[12] A. Rappaport. *Robots. Spiders and Crawlers. how Web and Int ranet search engines follow links to build indexes.* http://www.inktomi.com/prod ucts/search/support/docs/wp-spider/default.htm.

[13] R. Fielding. *Hpertext Transfer Protocol - HTTP/1.1.* RFC 2 616. June. 1999.

ftp://ftp.isi.edu/in-notes/rfc2616.txt

[14] S. Brin. L. Page. *The Anatomy of a Large-Scale Hypertextual Web Search Engine.* WWW7/Computer Networks 30(1-7): 107-117(1998).

[15] B. Krishnamurthy. J. Rexford. *Web Protocols and Practice. HTTP/1.1. Networking Protocols. Caching. and Traffic Measurement.* July 2001. ISBN 0-201-71088-9.

[16] *A Standard for Robot Exclusion.*

http://info.webcrawler.com/mak/projects/robots/norobots.html.

[17] D. L. Tennenhouse. J. M. Smith. W. D. Sincoskie. D. J. Wetherall. G. J. Minden. *A Survey of Active Network Research.* IEEE Communications Ma gazine. Vol. 35. No. 1. pp. 80-86. January 1997.

[18] A. Feldmann. *BLT: Bi-Layer Tracing of HTTP and TCP/IP.* WWW9 / Computer Networks 33(1-6): 321-335. 2000.

[19] A. Tomasic and H. Garcia-Molina. *Performance of inverted ind ices in shared-nothing distributed text document information retrieval systems.* Proceedings of the 2nd International Conference on Parallel and Distributed Inf ormation Systems. pages 8-17. January 1993.

[20] Apache Software Foundation. http://www.apache.org.

[21] S. Melnik. S. Raghavan. B. Yang. H. Garcia-Molina. *Building a Distributed Full-Text Index for the Web.* WWW10. pp. 396 - 406. May 2-5. 2001 . Hong Kong.

[22] B. Ribeiro-Neto and R. Barbosa. *Query performance for tightl y coupled distributed digital libraries.* Proceedings of the Third ACM Internati onal Conference on Digital Libraries. pages 182-190. June 1998.

[23] A. Broder. R. Kumar. F. Maghoul. P. Raghava. S. Rajagopalan. R. St ata. A. Tomkins. J. Wiener. *Graph structure in the web.* WWW9. 2000. http: //www9.org/w9cdrom/160/160.html.

[24] E. W. Zegura. K. L. Calvert. S. Bhattacharjee. *How to Mod el an Internetwork*. INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE C omputing Societies. Networking Next Generation.. Proceedings IEEE. Vol.2. 1996.

[25] P. Gburzynski. Protocol Design for Local and Metropolitan Area Ne tworks. Prentice Hall. 1996.

[26] Z. Liu. N. Niclausse. and C. Jalpa-Villanueva. *Web Traffic M odeling and Performance Comparison Between HTTP1.0 and HTTP1.1*. In System Perfo rmance Evaluation: Methodologies and Applications. August 1999. http://www.-sop. inria.fr/mistral/personnel/Zhen.Liu/Papers/wagon_perf99.ps.gz.

[27] H. Choi. J. Limb. *A Behavioral Model of Web Traffic*. Proce edings of the Seventh Annual International Conference on Network Protocols. 1999 http://users.ece.gatech.edu/ hkchoi/model.pdf.

Appendix A

The Ocaml code that implements all the services:

```
open Hashtbl
open Rwlock
open Activehost
open Indextbl
open Basis


(* Configuration constants *)
let index_store_size = 200         (* max num packets to keep track of *)


(* indexing table located on the router *)
let itbl: (Activehost.activehost, index_info) Hashtbl.t =
Hashtbl.create index_store_size

(* lock for the index table itself *)
let itbl_rwl: Rwlock.t = Rwlock.create()

let savehsh(mykey, dest) =
    try(
            with_read itbl_rwl
                 (function _ -> Hashtbl.find itbl dest) (); false
       )
    with Not_found ->(

            with_write itbl_rwl
                 (function _ ->
                      Hashtbl.add itbl dest {keynumber = mykey;
        dests = dest; rwl = Rwlock.create()}
```

```
                                ) ();
            true)


let get_index_hsh () =
    let indice = ref [] in
    with_read itbl_rwl
        (function _ -> Hashtbl.iter (function _ ->
                                    function ide -> with_read ide.rwl
                                    (function _ ->
                                         indice := (ide.keynumber, ide.dests)::(!indice)
                                    ()) itbl ) ();
    !indice

let delete dest =
        with_write itbl_rwl
                (function _ -> Hashtbl.remove itbl dest ) ()


let empty_indextbl() =
        Hashtbl.clear itbl
```

Appendix B

The Wrapper which maps service names to their corresponding functions

```
open Activehost
open Basis
open Eval
open Services
open PlanExn
open Indextbl
open Rwlock

(* interface function *)
let save_dest_hsh (p,l) =
        match l with
                [Key(h,i); Host dest] ->
                        Bool(Savehsh.savehsh((h,i),dest))
              | _ -> typecheck_args "saveDestHsh" l [KeyType; HostType]; Unit

let get_index_hsh(p,l) =
        match l with
                [] ->
                      VList (List.map
                            (function((h,i), ho) -> VTuple [Key(h,i); Host ho])
(Savehsh.get_index_hsh()))

             |  _ -> raise (ExecException(ArgMismatch("getIndexHsh", List.map
                                                  (function v -> Val v) l, 0)))
let delete_dest(p,l) =
        match l with
                [Host h] ->
```

```
                        (try
                                Savehsh.delete(h);
                                Unit
                        with Not_found ->
                                raise (PLANException "NotFound"))
                  | _ -> typecheck_args "delete" 1 [HostType]; Unit

let empty_indextbl(p,l) =
        match l with
                [] -> Savehsh.empty_indextbl(); Unit
              | _ -> typecheck_args "emptyIndextbl" l []; Unit

let register_svcs() =
    register_svc("saveDestHsh",save_dest_hsh,Some "(key, host) -> bool");
    register_svc("getIndexHsh", get_index_hsh, Some "void -> (key * host) list")
    register_svc("delete", delete_dest, Some "host -> unit");
    register_svc("emptyIndextbl", empty_indextbl, Some "void -> unit")
```

Appendix C

A PLAN packet that invokes service "saveDestHsh"

```
(* save the destination in the index table on the intermediate nodes *)
fun pp(s, where) =
(print(s);
 print(where);
 print("!\n")
)

svc generateKey: void -> key
svc saveDestHsh: (key , host) -> bool
svc getIndexHsh : void -> (key * host) list

fun savedest(dest:host) =
   (if (thisHostIs (dest)) then ()
    else   (
           let val key1 = generateKey()
               val nextdev = defaultRoute(dest) in
           (
                 (if(saveDestHsh(key1,dest)) then
                     OnRemote(|pp|("the destination has been saved on ",
 thisHostOf(getSrcDev())), getSrc(), getRB()/10, defaultRoute)
                  else OnRemote(|pp|("The same entry exists on ",
    thisHostOf(getSrcDev())), getSrc(), getRB()/10, defaultRoute) );
                      OnNeighbor(|savedest|(dest), fst nextdev, getRB(),
snd nextdev)
           ) end
          )
    )
```