# THESIS

HIGH PERFORMANCE CLUSTER-BASED WEB SPIDERS

**KASOM KOHT-ARSA**

GRADUATE SCHOOL, KASETSART UNIVERSITY

2003

# THESIS APPROVAL

**GRADUATE SCHOOL, KASETSART UNIVERSITY**

Master of Engineering (Computer Engineering)
**DEGREE**

| | |
|---|---|
| Computer Engineering | Computer Engineering |
| **FIELD** | **DEPARTMENT** |

**TITLE:** High Performance Cluster-based Web Spiders

**NAME:** Mr. Kasom Koht-arsa

**THIS THESIS HAS BEEN ACCEPTED BY**

.................................................... **THESIS ADVISOR**
( Associate Professor Surasak Sanguanpong, M.Eng. )

.................................................... **COMMITTEE MEMBER**
( Mr. Pirawat Watanapongse, Ph.D. )

.................................................... **COMMITTEE MEMBER**
( Assistant Professor Siriporn Ongroungrueng, M.S. )

.................................................... **DEPARTMENT HEAD**
( Mr. Pirawat Watanapongse, Ph.D. )

**APPROVED BY THE GRADUATE SCHOOL ON** ....................................................

.................................................... **DEAN**
( Professor Tasnee Attanandana, D.Agr. )

THESIS

# HIGH PERFORMANCE CLUSTER-BASED WEB SPIDERS

**KASOM KOHT-ARSA**

A Thesis Submitted in Partial Fulfilment of
the Requirements for the Degree of
Master of Engineering (Computer Engineering)
Graduate School, Kasetsart University
2003

ISBN 974-385-218-5

Kasom Koht-arsa 2003: High Performance Cluster-based Web Spiders. Master of Engineering (Computer Engineering), Major Field: Computer Engineering, Department of Computer Engineering. Thesis Advisor: Associate Professor Surasak Sanguanpong, M. Eng. 58 pages.
ISBN 974-358-218-5

Search engines and other web services primary rely on web spiders to collect large amount of data for indexing and analysis. Data collection can be performed by several agents of web spiders running in parallel or distributed manner over a cluster of workstations. This parallelization is often necessary in order to cope with a large number of pages in a reasonable amount of time. For this reason, design of high performance web spiders is a challenged task due to the large scale of the web.

There are two important aspects in designing efficient web spiders, i.e. crawling strategy and crawling performance. Crawling strategy deals with the way to prioritize documents for downloading. Meanwhile, crawling performance deals with the way to optimize spider performance. This thesis describes the design and implementation of "KSpider", a scalable cluster-based web spider under the performance aspect. Significant features of KSpider are scalability, robustness, flexibility and reconfigurability. KSpider can scale to download several hundred or thousand URLs per second without overwhelming any particular web server. It is substantially resilient against system crashes due to downloading and can be customized to other various web applications.

This thesis presents the KSpider's system architecture, discusses I/O performance issues, and proposes solutions to overcome OS limitations. In addition, several sophisticated designs enabling high performance are described in this thesis, such as workload distributions, in-memory URLs compression, phase swapping technique, scheduling policy, robust hyperlinks extraction, parallel data storage, and parallel DNS resolver. Finally, preliminary empirical results of data collection based on Thai web using KSpider are reported.

## ACKNOWLEDGEMENTS

I would like to thank to my parent for their supporting and everything. I would like to thank to the thesis committee, Assoc. Prof. Surasak Sanguanpong, Dr. Pirawat Watanapongse, Asst. Prof. Siriporn Ongroungrueng for their suggestions and encouragement. I would like to thank to Mr.Teerasak Rodchusang, the head of Networking Division at the Office of University Computer Services, for kindly supporting and co-operation on network configuration. I would like to thank to my friends and every one in the department of computer engineering for their supporting. This works is partially funded by the KURDI research grant SRU 2.43.

Kasom Koht-arsa
March 2003

## TABLE OF CONTENTS

## LIST OF TABLES

# LIST OF FIGURES

## LIST OF FIGURES (Cont'd)

# HIGH PERFORMANCE CLUSTER-BASED WEB SPIDERS

## INTRODUCTION

The World Wide Web (WWW) or web can be viewed as a huge distributed database across several million of hosts over the Internet where data entities are stored as web pages on web servers. Web pages are mostly unstructured or poorly structured documents and their logical relationships are represented by hyperlinks. Recent study in the year 2000 shown that "deep" web pages consists of about 550 billion pages (University of California, 2000) with the growth rate 7.5 million pages per day. Due to the enormous size of the web, search engines play more and more important role as a primary tool for locating information. Every search engine relies on massive collection mechanism called web spiders or crawlers or robots. These spiders "crawl" across the web, following hyperlinks from site to site, storing downloaded pages they visit to build a searchable web pages index. Most search engines compete against each other with the number of indexed pages, quality of returned pages, and response time. Even the largest search engines, such as Google (Google, 2003) and Alltheweb (Fast Search & Transfer ASA, 2003), can currently cover less than 0.5% of the publicly accessible web.

Search engines are one of the primary ways that Internet users find information. In summary, a typical search engine consists of 3 main components as illustrated in Figure 1.

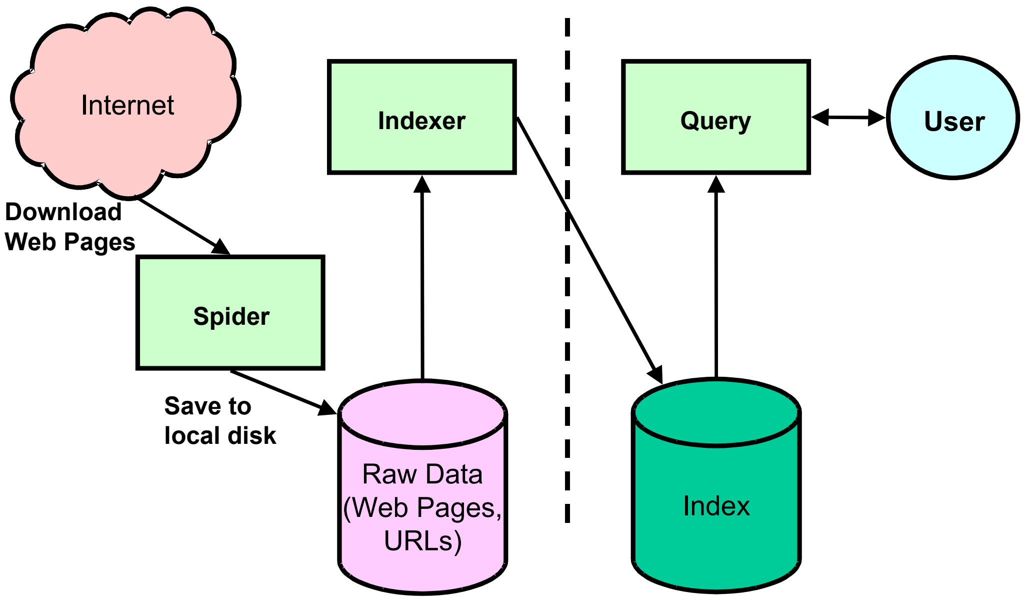


**Figure 1** Components of search engines

From Figure 1, the "Spider" is responsible for downloading web pages from the Internet and stores the data on the local disks. The "Indexer" uses the data collected from the spider to build an index for later query. The "Query" search the query from the user using the already built index.

There are two important aspects in designing efficient web spiders, i.e. crawling strategy and crawling performance. Crawling strategy deals with the way the spider decides to what pages should be downloaded next. Generally, the web spider cannot download all pages on the web due to the limitation of its resources compared to the size of the web. Recent works in this aspect concerns about how the web spiders select the pages and visit important pages first (Cho et al, 1998) (Najork and Wiener, 2001) including incremental crawling to detect page changes (Cho and Garcia-Molina, 2000) (Risvik and Michelsen, 2002). Crawling performance deals with the way to optimize spider performance. Each search engine has its own proprietary and highly optimized methodology. (Burner, 1997) (Sergey and Lawrence, 1997) (Heydon and Najork, 1999) Crawling can be performed by several agents of web spiders running in parallel or distributed over a cluster of workstations. This parallelization is often necessary in order to cope with a large number of pages in a reasonable amount of time. Besides the parallel choice, a good system and I/O design have tremendous impacts on spider's performance.

This thesis describes the design and implementation of *KSpider*, a scalable cluster-based web spider. The aspect of crawling strategy is beyond the scope of this work. In other words, the thesis emphasizes on high performance web spider based on parallel scheme, I/O and network efficiency, including storage space management aspects. The design and implementation of a web spider might seem to be simple. But it is not a case for large-scale web spiders, where several hundred of pages per second have to be downloaded. One could implement a very high-speed web spider, which has a capability to massively download web pages at a rate of thousand pages per second. However, without careful design considerations, this could create overloading problems on destination networks and often led to several phone calls or e-mail complaints from system administrators. In the worse situation, Intrusion Detection Systems (IDS) installed at destination networks may classify this incident as Denial-of-Services (DoS) attacks and will automatically block access from web spiders for security reasons. There are many underlie design concepts, which have guided the implementation of the KSpider. The important concepts are described as the following sections.

## Scalability

One of the most important objectives was to design a web spider whose performance can be scaled up by adding more additional nodes to the cluster. The key point is to create a completed distributed mechanism without any centralized node.

## Load Balancing

KSpider relies on the network workstations (also called *cluster*). In order to maximize performance of the nodes in the cluster, the workload must be distributed and balanced across multiple nodes.

## Fault-tolerance and Robustness.

Several issues arise in the aspect of fault-tolerance and robustness of the system. First, crawling in KSpider is completely distributed to nodes inside the cluster without centralize control. No single-point of failure exists in the system. Hardware crashes will not interrupt the whole system. Second, since web crawling often takes time from weeks or months, the system need to be able to tolerate to the network interruption. Emergency system interruption must be allowed without losing too much collected information. Resuming must be available to start from the failure point. Third, the system has to interact with a large amount of server; it has to tolerate with the strange behavior of servers. The web spider often faces the problem of "spider traps" which may degrade spider performance or rapidly exhausted the system resources. Some kinds of spider traps are an infinite number of URLs, and excessive page size generated by CGI scripts. KSpider is designed to handle spider traps in a considerably level. Pages or even entire servers may be excluded under some circumstances to preserve crawling performance, since a subset of the web can be downloaded anyway.

## Space saving

Large-scale web spiders are facing many challenge problems such as dealing with a large number of hyperlinks identified by the URLs (Uniform Resource Locators). The web spider must keep track of all the URLs it already visited. This is not a trivial task when a large amount of URLs must be handled. Keeping URLs on disk would degrade the system performance because of the high I/O latency. The KSpider is designed to handle all URLs in the central memory using exclusive compression technique. Moreover, dealing with the data from the web pages is also crucial. Several hundred million pages imply that there are terabytes of data to be managed. This could be accomplished by a special design of data structure and pages compression technique.

## High performance

It has been recognized that as the size of the web grows, single crawling process approach can not gather pages in a reasonable of time. Advanced spiders run with multiple processes in parallel as well as utilize distributed fashion to maximized download rate. The KSpider achieves several hundred pages per second on low cost, commodity of the shelve hardware.

## Etiquette

KSpider is designed to ensure that, while keeping a good parallelism, too many requests should not send to the same web server in a short period of time. The phase swapping technique is used to minimize synchronization between workstations in the cluster, while also reduce the working set of the URLs. It also ensures that no spider from any different node in the cluster is going to contact to the same set of web servers. KSpider's design and implementation is primary concern about the performance, meanwhile maintaining acceptable politeness according to the robot exclusion standard (Koster, 1993 a).

## Flexibility and Reconfigurability

There is naturally trade-off between performance requirements and the system optimization (politeness, space saving, and scalability) in web spider design. Concurrent supporting all of these features is a challenge design issue. KSpider is specially designed to maximize the performance under Linux operating system. Several issues are discussed based on the Linux operating system. However, standard tools and library are used so that it can be deployed in other UNIX's platform.

The spider is highly configurable to match the available hardware and the running environments. For example, one can use multiple disks with different size to store web pages; the number of data collector threads can be configured to match the network and the server's resources. Other parameters that control KSpider's behaviors can be adjusted for an appropriate environment via configuration files.

## Problems

There are many challenges in designing a high-performance large-scale web spider. In this section, those challenge problems are addresses with their brief solutions.

### 1. URL Size

The web spider needs to search and retrieve URLs at all times. The way to store, search, and retrieve URLs require a very fast technique. The secondary storage such as hard disk has its low order of efficiency in nature. For a fast disk today, about 5 milliseconds are needed for the drive to seek for a data. This mean that a disk can perform about $1/(5x10^{-3}) = 200$ seek per one second. Let the average number of URL per page is equal to 13 (Sanguanpong et al, 2000). For a web page with about 13 links to be processed, a disk sub system can perform only about 200/13 = 15.4 URLs per second.

Handling all URLs in the central memory would maximize the efficiency of URLs access. A recent reports has shown that the size of URL is about 59.5 bytes in average (Sanguanpong et al, 2000). For example, to store 100 million URLs, about 5.5 Gbytes of memory is required. Several operating systems are not able to handle a large amount of data direct into the memory. In Linux operating system, only 2000 Mbytes of contiguous virtual address space is available.

### 2. Data Size

The data from 100 million URLs is regarded as a large size of data. Storing 100 million pages with an average of 10 Kbytes of size (Sanguanpong et al, 2000) requires about one Terabytes of disk space. Multiple disk drives are demanded to hold such data. Tradition array of disk such as RAID system is inadequate to reach the maximum performance. A design of data partitions across multiple disks and appropriate data compression scheme must be highly considered.

## 3. Robot Exclusion Standard and Netiquette

The RES or Robot Exclusion Standard (Koster, 1993 b) is a simple mechanism that the web administrators inform a spider what should and should not be examined. The word "should" means that RES is only guidance for spiders. There's no method that a publicly accessible site can enforce robot exclusion, except keeping a list spider's IP addresses and explicitly blocking them at the web server or at a firewall. However, it is extremely important for web spiders to respect the RES, which is represented by a file "*robots.txt*" in the root of the web site. The web spider has to check every URL specified inside the file to ensure that it did not download any disallowed pages. The spider should keep a copy of the file from each web site to avoid reloading it repeatedly. Moreover, when the web spider downloads pages from the web, it consumes disk and CPU resources of the web servers. The web spider should minimize its impact on these resources on remote web servers.

## 4. DNS Resolver

The web spider needs to resolve a host name into IP address using the DNS. The DNS Resolver on the GNU C Library (Free Software Foundation, Inc. 2002) (glibc) has no caching mechanism. It needs to send a query every time when it resolves a host name. Moreover, the resolver contacts only a primary DNS server at a time according to the client configuration at the host the resolver resides. Although the secondary DNS server can assume the duties of the primary in the event the primary fail, the duration spent waiting for timeout of the primary DNS server will greatly impact spider's task. KSpider solve this problem by adding internal DNS caching and modify the resolver to allow it to contact several DNS servers in a more sophisticated way.

## 5. Spider Traps

Spider traps are the web pages that generate an infinite number of URLs, or have an unusual property that may crash a web spider. Detection and filtering modules will handle such spider traps. Here are some kinds of spider traps.

### 5.1 Symbolic Links

A hard link and a symbolic link (also called softlink) within a file system can create a directory cycle and it would be classified as a type of spider traps. A directory cycle arises in a directory tree when an entry in a directory points to its ancestor directory. For example, if a directory "image" has a symbolic link named "loop" which refers to itself, every file in this directory could be accessed under several names such as "*image/myfile.jpg*", "*image/loop/myfile.jpg*", "*image/loop/loop/ myfile.jpg*", and so on. Moreover, directory cycles can also occur in another complicated ways. A spider that follows a hyperlink contained the directory cycle will traverse the same directory tree and often downloads duplicated pages indefinitely. To prevent getting "stuck" in such endless loop, KSpider performs comprehensive cycle detection.

### 5.2 Excessive Page Size

Some page may have a very large or infinite size, such as some malfunction CGI pages. A web spider might try to download that page until it has exhausted its memory or disk space.

### 5.3 CGI Generated Pages

If not carefully use, a CGI can generate an infinite number of pages. Any pages of them may contain links to the other CGI generated pages.

## 6. <u>Network Utilization</u>

The Internet network connection is like a mesh that there are many paths leading to many servers around the world. A web spider should have a mechanism to distribute the workload to many network paths as much as possible.

## 7. <u>Linux's limits</u>

The Linux-kernel has some limitations that place some constraint on the designing of the web spider's architecture. The following issues address these limitations and some considerations as well as solutions to achieve the maximum efficiency.

### 7.1 Descriptor

A descriptor or a file handle is an integer value that a program uses to interact to a kernel. It is use to read or write to an open file or network socket or the other device. Every TCP socket, UDP socket and any open files uses a descriptor. Without a modification, the kernel and glibc library limit maximum number of descriptor a process can occupy to 1024. A web spider must take care of any file or sockets it wants to keep open not more than this limitation.

## 7.2 Local port range

For the outgoing TCP connection, the Linux-kernel allocates a source TCP port from the system's pool. If the pool is exhausted, no connection can be further established. The ports are returned back to the system pool after the connection is completely terminated. When the web spider repeatedly connects and close connection, many sockets may remain in TIME_WAIT state and then the local port pool can be exhausted easily. The simple solution is to extend the range of local port pool.

## 7.3 Memory Map

A process has its limitation to utilize the central memory. On x86 system, a Linux's user process makes use of 32 bits linear memory addressing; hence $2^{32}$ or 4 Gbytes of memory can be addressed as shown in Figure 2. The Linux kernel occupied the upper 1 Gbytes address space to map the physical memory, so there are 3 Gbytes remained for the user process.

The lower 3 Gbytes itself is divided into many parts. The first 128 Mbytes is not mapped or allocated but reserved for system proposes. It is useful to catch any attempt to de-reference the NULL pointer. Linux currently uses the ELF (Executable and Linking Format) which is a binary format originally developed by UNIX System Laboratories (see http://www.linux.org/docs/lpd/howto/GCC-HOWTO/). Under this binary image scheme, the Linux's program is usually mapped at 0x08048000. The shared libraries or any memory mapped files start at 0x40000000. The stack grows down from 0xc0000000, occupied 8 Mbytes by default.

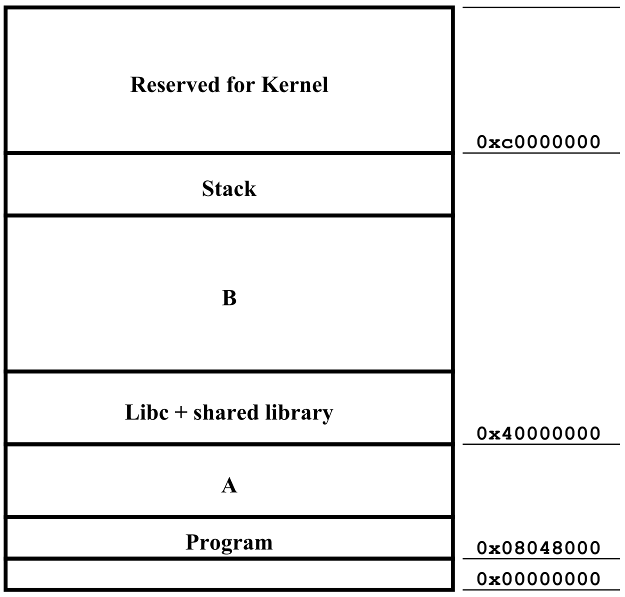


**Figure 2** Linux Process Memory Layout

Under this memory layout, a process can use a memory-mapped file or a dynamically allocated for about 850 Mbytes in region A, and about 2000 Mbytes in region B.

In a multi-threaded program, all threads share the same virtual memory address space; hence the maximum memory the process can have is still about 2850 Mbytes. Every thread also must have its own stack, when using a large number of threads, large portion of virtual address space is required. Default of 8 Mbytes per thread yields that only about 2000/8 = 250 threads can be created.

## 7.4 Virtual Memory

Linux on x86 systems limits about 3 Gbytes of the maximum virtual memory size a process may. This means that even though a machine has more than 3 Gbytes of memory or swap spaces, any process on that system will not be able to use more than 3 Gbytes. A fraction 3 Gbytes of virtual memory is already in used; hence a user program has less than 3 Gbytes of memory left for utilization. This number must be highly considerable for the program design.

# LITERATURE REVIEWS

Research and development of web spider has its milestone less than a decade (Eichmann, 1994) (Pinkerton, 1994). Unfortunately, research relevant to the design of practical large-scale web spiders has not been publicly described due to the competitive nature of search engines business. The only system described in detail in the literature appears to be the Google crawler (Sergey and Lawrence, 1997), Internet Archive (Burner, 1997) and Mercator system (Heydon and Najork, 1999).

Compression is one of the important aspects in KSpider to handle a large amount of URLs. The Internet Archive solves the excessive number of URLs with a giant bitmap approach. A chunk of memory is allocated for a bitmap, and then initialized it to be zero. For any URL, ten hash values are computes using ten different hashing algorithms. These hash values are checked against the bitmap. If there are any bits in the location pointed by those hash values that is not set, this URL has not be seen before. All the location pointed by the hash values are then set. This method relies on the hash function that may produce a "false positives." Furthermore, the method can't fulfil the search engine's requirements to retrieve the URLs back.

The Adaptive Web Caching project (Michel et al, 2002) uses a simple URL table compression that is based on a hierarchical URL decomposition to aggregate URLs sharing common prefixes and an incremental hashing function to minimize collisions between prefixes. While this method is good for constructing a forwarding table for the web cache to locate the nearest copy of a requested URL's contents, it's too coarse to be used with a search engine or a web spider. It lacks the ability to reconstruct the actual URL (needed by search engines) and may also produce false positives.

The Connectivity Server (Bharat et al, 1998) stores the URLs by sorting them lexicographically. The URLs are compressed using a delta encoding, a URL is stored using the difference (delta) between the current and previous URL. Since the common prefix between two URLs from the same server is often quite long, this scheme reduces the storage requirements significantly. This method requires that URLs have to be sorted first. After the compressed URLs have been constructed, no URL can be later added. The second version of the Connectivity Server (Wickremesinghe et al, 2000) has significantly improved the compression rate to store linkage information for approximately 400 million pages in 8 GB of RAM. However, this method is suitable for construction of the web graph from whole-prepared URLs at once. Furthermore, there is no way to search for the existence of any given URL, except to scan through the list of URLs. Hence, it is not suitable for web spiders such that a new URL is continuous generated on the fly.

The Mercator uses checksum of the URLs to check for duplication. The checksums are computed using Rabin’s fingerprint algorithm (Rabin, 1981). The checksum are stored on disk, slightly over 5 GB is required to store checksums of 1 billion URLs. To reduce the disk access, the in-memory cache of $2^{18}$ entries is used to keep the popular URLs. The Mercator uses synchronous I/O with multithreads, and runs on single machine.

# MATERIALS AND METHODS

## Materials and Environments

### 1. Hardware

Four set of Athlon XP 1500+ CPU with 768 Mbytes DDR DRAM, six of 35 GBytes, 7200 RPM SCSI Harddisk and an Intel E1000 Gigabit Ethernet interface card. They are connected together using 3com gigabit Ethernet switch.

### 2. Software

The operating system and the system software is RedHat 7.3 with Linux kernel version 2.4.18 and GNU C++ version 2.96. The LZO data compression library version 1.08 is used as the compression library.

### 3. Internet Connection

All four machines are connected together using the gigabit Ethernet switch. The gigabit switch is connected to the campus network's backbone using four fast Ethernet links. The campus network is connected to the Internet through two 155 Mbps ATM links. The first ATM link is connected to UniNet, which is an inter-university network. The second ATM link is connected to NECTEC. The network map of Thai Internet connectivity is available at http://www.ntl.nectec.or.th/internet/map/current.html. UniNet connects to global Internet with 155 Mbps full-duplex bandwidth, and peering with Abilene (Internet2 Backbone) at 155 Mbps bandwidth and to local Internet Exchange at the Communications Authority of Thailand (TH-NIX) with 68 Mbps full-duplex bandwidth. UniNet has serverals gateways in Bangkok, which connect each other with 155 Mbps. The regional nodes connect to the gateways with 34 Mbps. The access nodes in different province connect to the regional nodes with varying bandwidth from 256 Kbps to 2 Mbps. The overall network connection diagram is shown in Figure 3.

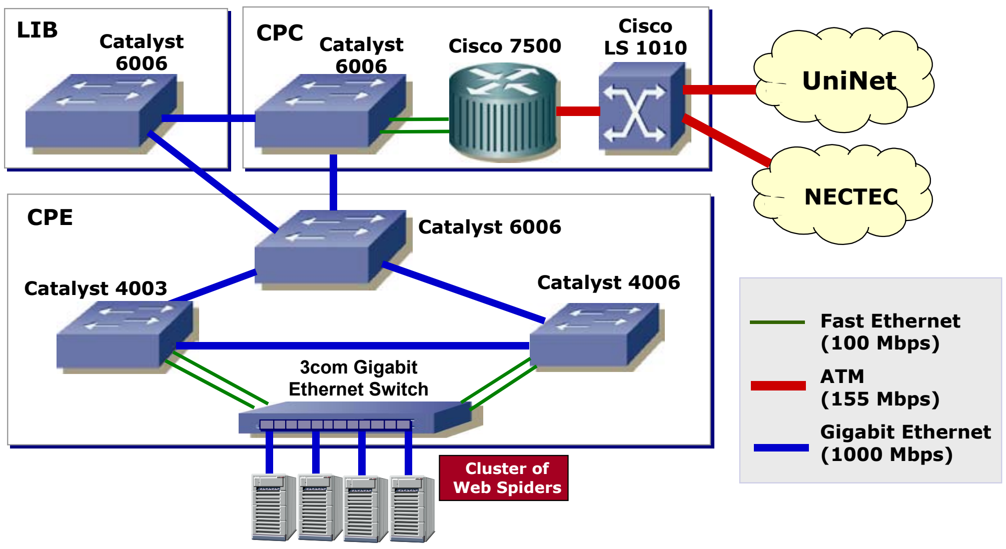


**Figure 3** Internet network connection

The above system is not prepared solely for crawling purpose but for search engines and other database projects. Results of crawling, which will be described later, shown that KSpider is a CPU and memory intensive system. Experimental pointed that for the crawling task; network connection for each node could be handled using the Fast Ethernet instead of the Gigabit Ethernet.

## Methods

### 1. Cluster

The web spider based on Beowulf cluster (Sterling et al, 1995). The cluster is a group of workstations that are connected together using a high-speed network interconnection. The cluster is connected to the Internet through the campus network. The overall architecture of KSpider is shown in Figure 4.

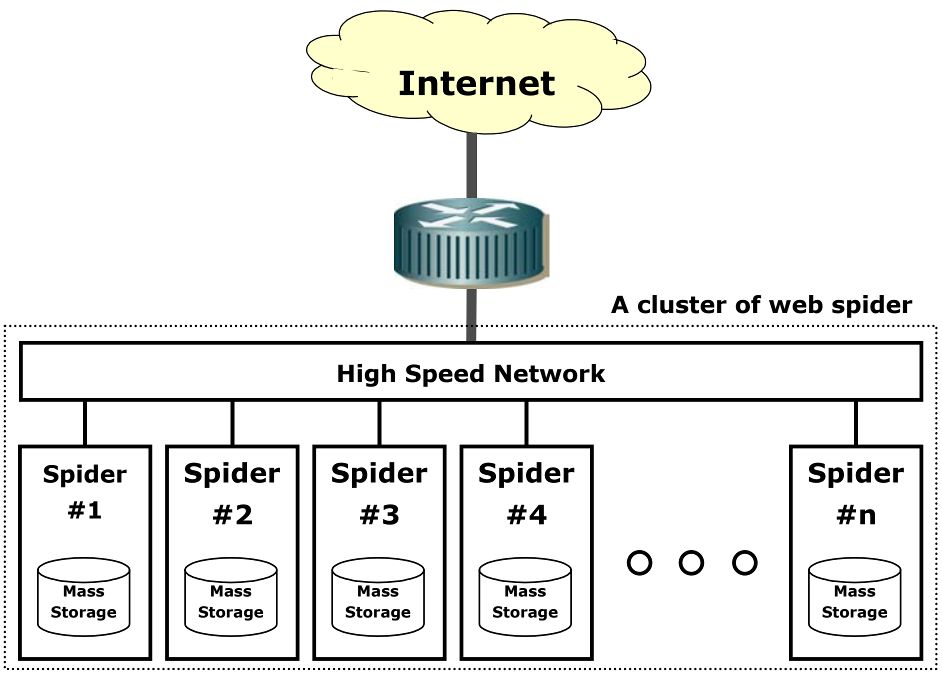


**Figure 4** Overall syste m architecture

## 2. Architecture

Fetching data through the Internet usually take time. The network might be congested, the server might be busy, and the latency might be high due to the number of hop counts along the network path. Fetching data in parallel, either in the same machine or in the cluster, can minimize some of these problems. While parallelizing the fetching method can gain more performance, a special care must be taken to ensure that the spider will not overload the network or the web servers.

Like any parallel program, every node (Spider #1, spider #2, …, spider #n) in the cluster needs to communicate and synchronize to each other. Every node has to exchange the URLs belonging to other nodes. KSpider has no central node, so every node does the same task independently but with a different set of data. The distribution scheme of the data set is describes later in the next section. The web spider also needs to make sure that each node would not fetch the data from the same web server at the same time. To achieve this condition, a technique called 'phase swapping' is proposed and it will be described in section 4. To efficiently keep track of the URLs in the memory without disk access, a compression algorithm is described in section 7.

KSpider is implemented in C++ on top of Linux operating system. It consists of five main components as shown in Figure 5. Each component is described according to the step how KSpider works as the following.

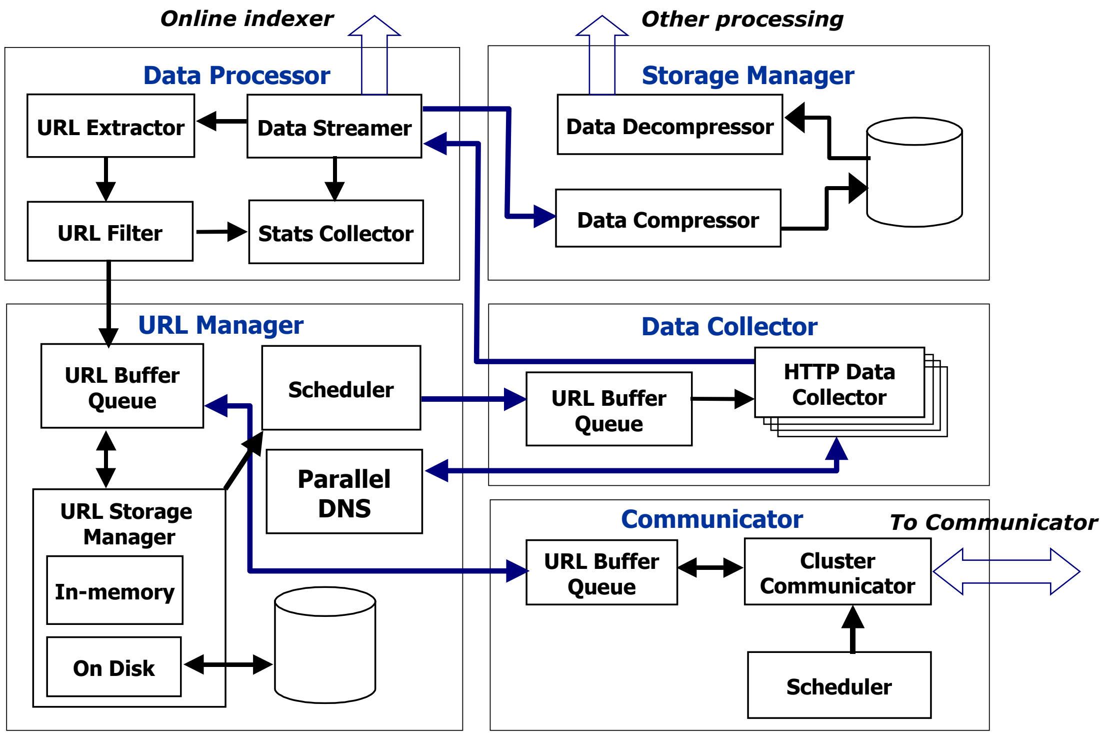


**Figure 5** The architecture of KSpider

### 2.1 URL Manager

The URL manager is responsible on all about URL handling. Each URL Manager in a node keeps track of a smaller disjoint of subset, compared to the other nodes. The URLs are enqueued in the URL Buffer Queue, and then they are filtered using a regular expression. After that, the filtered URLs are compressed and stored in memory for further processing. The scheduler on the URL Manager will select and schedule the URLs to fetch by sending the list of URLs to the Data Collector.

The scheduler inside the URL Manager chooses a URL to fetch from the active URLs set. The URLs of the same web server are packed together in a list. Packing the list of URLs enables the data collector to utilize HTTP1.1's persistent connection (Fielding et al, 1999) and pipelining to gain more performance (Nielsen et al, 1997). The persistent connection and pipelining reduce the overhead of the connection time an also reduce the network traffic. The number of URLs inside a list should not be too large, for a good distribution. The number may be between X and Y. In the current implementation, it is packed with 20 URLs per list. Further details about this implementation are described in section 6.

### 2.2 Data Collector

The Data Collector enqueues the URL list sent from the URL Manager. It takes care of collector threads to fetch the data from the web servers. The collector threads get a list of URL from the queue and send the request to the web server using

HTTP/1.1 persistent connection/pipelining. The fetched data will be passed to the Data Processor for further processing.

### 2.3 Data Processor

The Data Processor sits between the Data Collector and the storage manager. Any action that needs to be performed on the web data is taken there. There are: hyperlinks extraction, statistics collection, URL filtering, and etc. At this stage, the downloaded pages may be indexed online by search engine's indexing module to make searchable database for search engine purpose. After that the data are passed to the storage manager. The fast and robust 'URL Extractor' takes the responsibility for hyperlinks extraction. It is very tolerate to many kind of mal-format HTML pages. Section 9 contains the details of the URL Extractor.

### 2.4 Storage manager

The storage manager compresses and decompresses, storing and retrieving the data. The data from the web pages are compressed and then packed together into a large file. The LZO algorithm (Oberhumber, 1996) is used as the compression library because it is very fast, while the compression ratio is acceptable. The details of this part are in section 8.

### 2.5 Communicator

For any given URL, each node has prior knowledge to handle URL locally without communicating. Whenever, a new URL is extracted and the node found that it is not responsible for such URL, the communication is required to transfer that URL. This is the task of the Communicator to delivery URLs using UDP to another node in asynchronous fashion.

Obviously, sending one URL per one UDP packet is not an efficient way due to the protocol overheads. Each UDP packet encapsulated in an Ethernet frame has a constant overhead of 42 bytes (14 bytes Ethernet header + 20 bytes IP header + 8 bytes UDP header) to transport to IP network. In order to minimize the overhead, several URLs are packed together and sent out within a UDP packet. The Communicator has a buffer to hold several URLs before sending. It carefully accumulates the URLs and bewares that the total URLs size must not create IP fragmentation.

In the case that only few URLs are available for a quite period of time (e.g. when the spider has just started working), the Communicator will periodically send the packed of URLs out. To provide reliability mechanism, the scheduler of the communicator takes care of timeouts and data retransmission.

## 3. Data distribution

Data (web pages, images, and other file type) downloaded from the web servers are distributed over the nodes in the cluster. For any given URL, there is only one node that is responsible to fetch and keep data reference by that URL.

Let *Nnode* be a total number of nodes inside the cluster. Each node is assigned with a unique identifier, starting from 0 to *Nnode*-1. A simple hash function based on the summation of every character in the URL is used to distribute the URLs among the nodes in the cluster. When a node found a new URL, it computes a hash value of the URL using the hash function as shown in equation (1). The result is the number to indicate which node in the cluster has to handle that URL. The hash function guaranteed that every node has a disjoint set of the URLs.

$$hash(url) = \sum_{i=0}^{i<len(url)} url[i] \bmod Nnode \tag{1}$$

To better clarify this concept, suppose there are five nodes in the cluster and let $S_N$ denote the set of URLs belonging to node *N*. The hash will logically separate the whole URLs into five disjoint sets. Each set is represented by a column of $S_N$, (from $S_0$ to $S_4$) as shown in Figure 6.

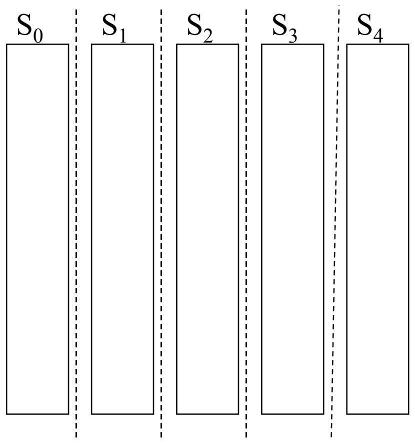


**Figure 6** Disjoint set o f the URLs (5 nodes)

## 4. Phase swapping

Refer to the sets of URLs distributed to the nodes in Figure 6, each node will have of sequence of URLs belong to the same server. Hence, it is likely that every node may download web pages from the same web severs at the same time. Should there are several nodes running simultaneously, it would increasingly generate heavy loads on destination servers. To prevent this problem, a sophisticate and efficient technique called 'phase swapping' is proposed. This technique does not only prevent the overload of the web servers, but also largely reduced the number of URLs the spider has to manage in any given time. The underlie concept of phase swapping is to group the URLs belonged to same web servers together and let each node works on the different set of servers at a time. After a pre-defined constant period, every node synchronously swaps the ‘working phase’ to a new set of web servers; hence it is called phase swapping.

A phase or a group of web servers can be created with hashing functions. Refer to the set of URLs from Figure 6; they will be hashed again using another simple hash function based on the summation of every character from the host name portion of the URL, as shown in equation (2). The bound of this second hashing function can be selected as appropriate and regarded as a number of phases allowed in the swapping. From the equation (2), the *Nphase* is a total number of phases.

To prevent the web spider from fetching data from the same web server at the same time, a rather complex but efficient technique called ‘phase swapping’ is proposed. This technique does not only prevent the overload of the target web servers, but also largely reduced the number of URLs the spider has to manage in any given time.

Refer to the set of URLs hashed from Figure 6, it has been hashed again using another simple hash function based on the summation of every character in the host name portion of the URL, as shown in the equation (2).

$$hash(url) = \sum_{i=0}^{i<len(url)} host(url)[i] \bmod Nphase \qquad (2)$$

Suppose that the number of phases is designate to be 6 (*Nphase*=6), hence the second hash function will splits each column of the URLs from Figure 5 into 2-dimensional view as shown Figure 7. Each row represents different phases (set of web servers). The notation $P_K$ denotes the set of URLs in the phase *K*.

At any given time, only one set (highlighted) is active on any given node (column), as shown in Figure 8. In any row there can be at most only one active set too. This means at any given time there will never be any node working (fetching) in the same set of web servers.

Note that the number of phases must be equal to or greater than the number of nodes. If the number of phases is less than the number of nodes, some node will fetch from the same set of servers. From the experiments, the optimum number of phases should be equal to the number of nodes in the cluster. Excessive number of phases results in the unbalanced of workload (see the experimental result in the later section).

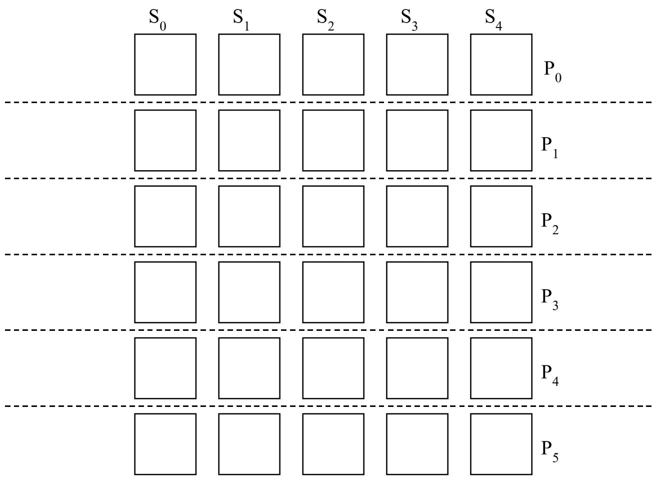


**Figure 7** Disjoint set of the URLs further split by the second hash function

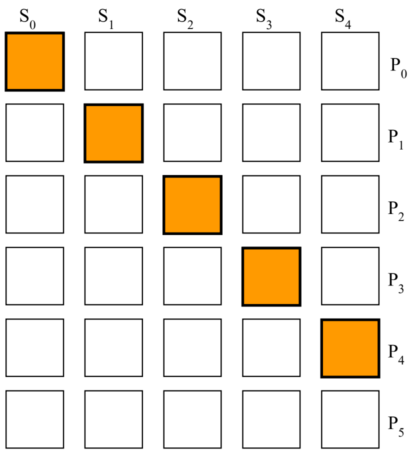


**Figure 8** The active set of URLs

Initially (see Figure 8), the node $S_n$ is assigned to handle $P_n$, i.e, there are working set of ($S_0$, $P_0$), ($S_1$, $P_1$), ($S_2$, $P_2$), ... ($S_N$, $P_N$) where $N$ is the number of nodes and $N \leq Nphase$. The active sets are changed by the system's wall clock, as shown in Figure 9. Let TM be the time of phase swapping. When the first phase swapping occurs (switching from $T_0$ to $T_1$), the active set of URLs from every node are switched to the next phase from ($P_K$ to $P_K$+1). When a node is active in the last phase $P_K$, where $K$=$Nphase$, the next phase is switched back to the first phase $P_0$, i.e., phase is rotated in a round-robin fashion.

In order to let the phase swapping in every node occur at the same time, every nodes need to synchronize the timing. KSpider utilizes the Network Time Protocol (NTP)(Mills, 1991) for the phase synchronization. The NTP is the standard time synchronization protocol that is efficient and very accurate. There is no master node or any implicit synchronization between nodes for the phase swapping.

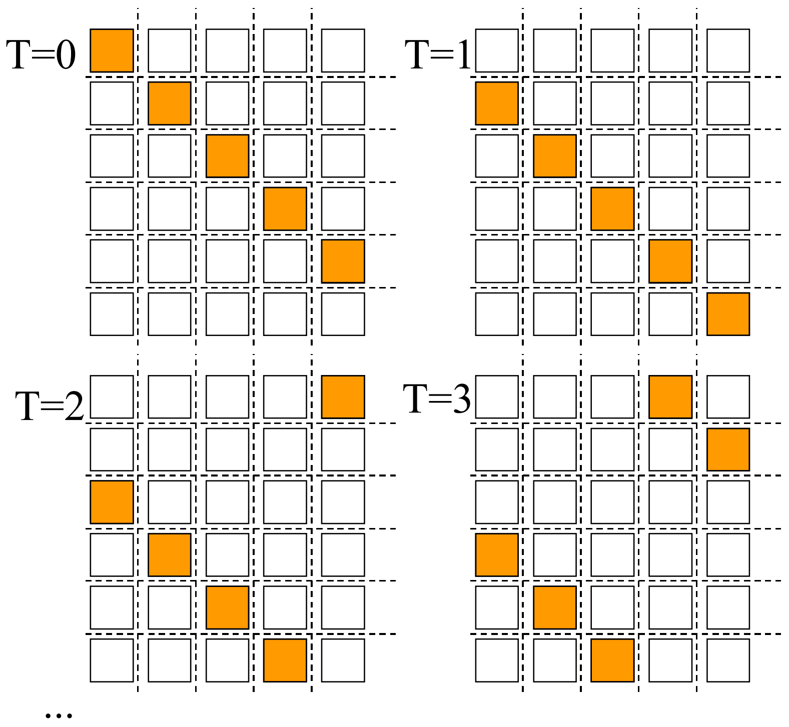


**Figure 9** The active set of URLs are changing as time changed

## 5. Parallel DNS Resolver

The Domain Name System (DNS) is used for resolving host name into IP address. Before a spider can create any connection to the remote web server, it has to contact DNS server to translate the host name into IP address. A large number of DNS queries will be generated due to several numbers of web servers the spider has to contact. In Linux, a function 'gethostbyname' in the glibc library is used. This function is not reentrant, i.e., it can not be run concurrently. There are 'gethostbyname_r' and 'gethostbyname2_r' variants that are reentrant, but none of them have a cache. They also lack the mechanism to specify time-out and to distribute loads between the DNS servers.

KSpider has been design to integrate the DNS caching mechanism to the resolver. This helps reducing DNS server workload when several thousand of hostname must be resolved in a short period of time. The custom-made DNS resolver in KSpider consists of two parts. The first part is a DNS cache. The cache is a hash table; each entry contains a small array that holds the server name and an IP addresses associated to that name. The host name is hashed using a simple hash function, as illustrated in equation (3). The replacement policy in the array is FIFO.

$$hash(hostName) = \sum_{i=0}^{i<len(hostName)} hostName[i] \bmod nBucket \tag{3}$$

## 6. Scheduler

The Scheduler plays an important role in this since it orders the execution of data downloading. It selects URLs from a set of URLs in the active phase and sends them to the Data Collector. URLs to be selected are in the compressed form of AVL tree as described before. Logically, the scheduler models all URLs as an array, which its index is used to access the URLs in the AVL tree. This array grows corresponded to each new encountered URL. The Scheduler takes each URL by scanning and uses array's index number for referencing to a corresponded URL. Scanning range is controlled by a sliding window, *sWin*. The window size, which is adjustable, is currently set to 250000 entries. This access method is quite similar to the FIFO queue. However, the size of array is always increased according to the number of URLs. No element is destroyed along the operation.

The structure of array is very simple. It is the array of integer value, called the array of URLStates as illustrated in Figure 10. After the Scheduler has already picked each URL out from the array, it assigns the state to that URL. There are four possible states of URLs as the following:

STATE_IN_LIST: Any newly discovered URL is in this state. It means that the URL is now in the list of URLs, but has not been yet scheduled for downloading.

STATE_IN_PROGRESS: Any already scheduled URL, but it has not yet complete downloaded, will be assigned to this state.

STATE_COMPLETED: Any already downloaded URL either successful or failed will be assigned to this state.

STATE_STALLED: Any scheduled URL that can not be downloaded due to server timeout.

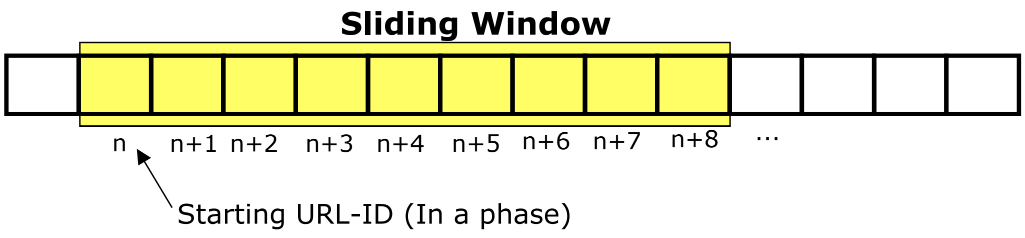


**Figure 10** The URLStates array

URLs selected from an array are not directly sent to the spider's threads. But they will be cached in a special buffer called a Bucket. The structure of Bucket is a two-dimension array as shown Figure 11. A URL will be pushed into the bucket using hashing function of host name. The hash function is shown in equation (4). Hence, every element in the same bucket's column is in the same set of host names. (The two hash functions use to split the URLs into phases and into the bucket are nearly identical, except the divider). The number of column (*nColumn*) is set to 1999 in the implementation. Its appropriate range should be between 199 to 4999. It is important to use a prime number that is not equal to the number of phases; otherwise the bucket will not be balanced.

Note that the number of column is obviously less than the number of existing host name; hence hashing collision can not be avoided. Each row of the Bucket is used to handle hashing collisions. In the implementation, the number of row is set to be 8. The optimum range should be from 4 to 16.

Each Element in the Bucket is not a simple data type but a complex structure of three values, i.e. List, Hostname and State. The List is an array of URLs which all belong to the same web servers denoted by Hostname. The State is an integer value used to denote the status of Bucket's element (to be described later).

For the sake of clarity, it is worth to give the operations based on such objects again. The scheduler selects a URL and hashes it using the equation (3). Each hashed URL will be put into the corresponded Bucket's Element. The number of URLs in the List of each Element independently increases until the List is full. Then, the URLs in the List are immediately sent the Pool of Collector for downloading. The purpose of Bucket is, therefore, very like the phase separation, i.e. to group the same URLs belonging to the same server until it reaches a fair amount of number (20 in the implementation). The benefit of this method is to reduce the number of connection sent to the web servers. Once there are sufficient number of URLs belonging to the same server, the HTTP persistent connection is fully utilized. That is, only one connection is created to the web server for downloading several URLs. Moreover, URLs are spread out randomly to the Bucket, waiting to be scheduled randomly with fewer overheads than other method liked sorting.

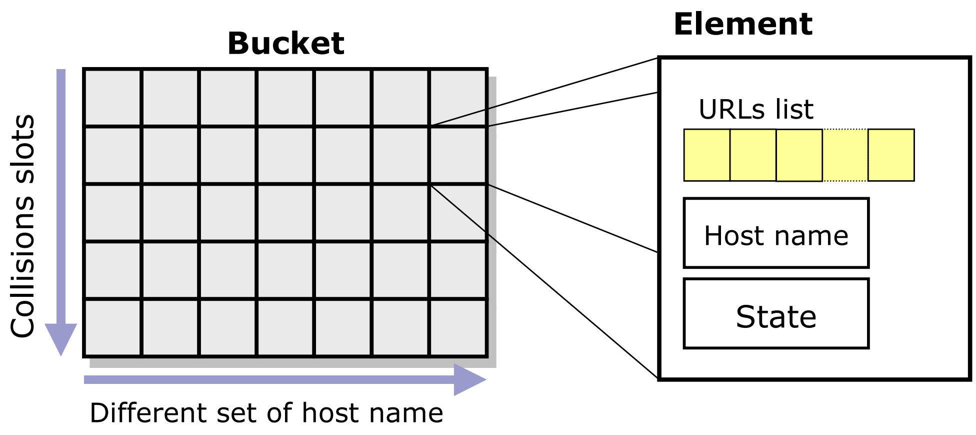


**Figure 11** Bucket and its element

$$hash(hostName) = \sum_{i=0}^{i<len(hostName)} hostName[i] \bmod nColumn \tag{4}$$

The last part of the Element, which has not been described yet, is the State. It is used to indicate the status of Element's processing. There are four possible states: 'STATE_FREE', 'STATE_QUEUING', 'STATE_FETCHING', and 'STATE_TIMEWAIT'.

The state 'STATE_FREE' indicates that the element in the Bucket is free, the Scheduler can store any URL from any host to it. Once the Scheduler stores a URL into any element, it will change the state to 'STATE_QUEUING' and change the host name according to the host name part of the URL. The Scheduler will also set the URLStates to indicate that this URL is in the 'InProgress' state.

The state 'STATE_QUEUING' indicates that this element in the Bucket is in enqueuing, i.e. waiting for more URLs to be put into.

If the list becomes full, the whole list is sent to the Data Collector, and the state will be changed to 'STATE_FETCHING'.

When the Data Collector has finished the fetching, it will notify the URL Manager to set the state in the element from 'STATE_FETCHING' to 'STATE_TIMEWAIT'. This state will prevent the spider to continuously fetching from the same server. The Scheduler will periodically reset any 'STATE_TIMEWAIT' to 'STATE_FREE'. The state diagram of the State of the element in the bucket is shown in Figure 12.

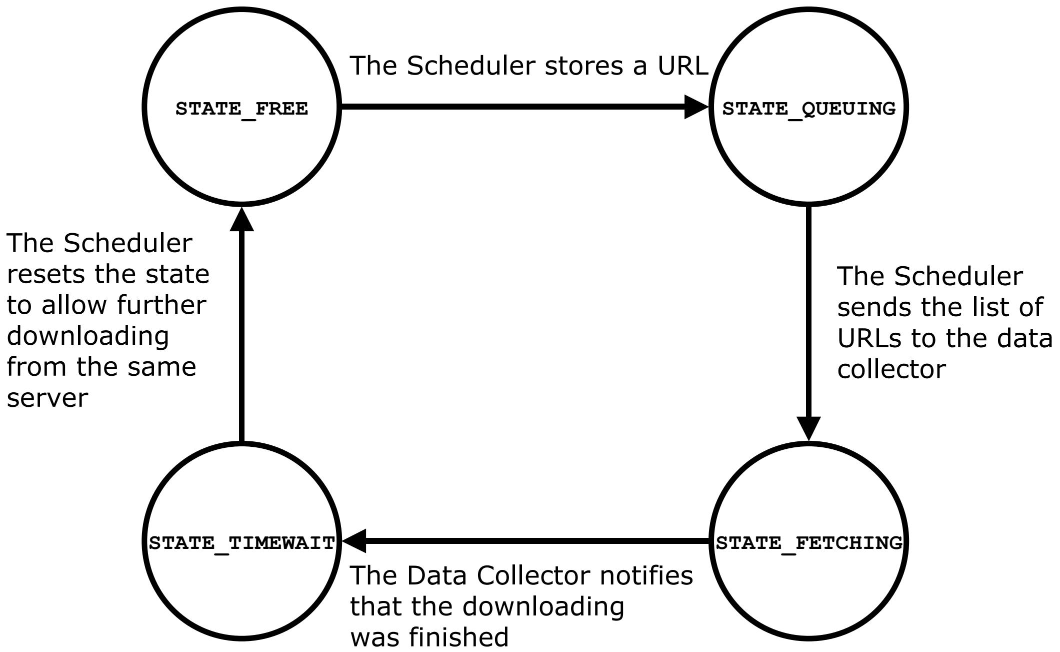

STATE_FREE
The Scheduler stores a URL
STATE_QUEUING
The Scheduler sends the list of URLs to the data collector
STATE_FETCHING
The Data Collector notifies that the downloading was finished
STATE_TIMEWAIT
The Scheduler resets the state to allow further downloading from the same server

**Figure 12** Cycle of the State in the Element of Scheduler's Bucket

## 7. URL compression

Storing, searching and retrieving a large number of URLs are one of the problems in designing large-scale search engines and web spiders. Search engines use the URLs to show the web page relevant to user queries, while spiders have to keep all of them in order to check whether a URL has been visited. To store 100 million URLs, with the average size about 59.5 bytes, it would require at least $59.5 \times 10^8$ bytes (approximately 5.5 GBytes) of storage space.

While system processor and memory speeds have continued to increase rapidly, application performance is still constrained by slow disk and I/O speeds. Most disks still have an average seek time about 4 to 12 milliseconds. It implies that the disk can be accessed randomly only about 80 to 250 times per second. This is one of the performance bottlenecks of search engines and web spiders for storing and retrieving of URLs.

Since the memory is many orders of magnitude faster than the hard disk, storing URLs in memory results in performance improvements. However, storing full URLs in memory is impractical because there are only few machines that offer a big size of main memory. Furthermore, most 32 bits machines/operating systems have a maximum virtual memory that a process may have about one to three gigabytes. To overcome this circumstance, a compression method is needed for reduction the size of URLs.

Like the delta encoding of the Connectivity Server, KSpider compresses the URLs by only keeping the differences of URLs tails. To find the URLs different, the sorting of URLs is required. However, a new URL is discovered on the fly and it is impractical to sort out the URLs list every time a new URL is created. Curtain data structure for efficient URLs keeping is required. One of an appropriate is the AVL tree.

An AVL tree (Sedgewick and Flajolet, 1995) is a special kind of a binary search tree. It is a height balanced binary search tree that derives a property of a binary search tree. For any given node, the predecessor or the successor of that node must be one of its root node, or the maximum node of the left child, or the minimum node of the right child. Finding the predecessor/successor of a newly added node is much easier, because they must be one of its root node that it travel pass. That is because before the rotation of an AVL tree, the newly added node has no child.

A node structure is illustrated by like in Figure 13. Each node in the tree contains five fields. The “RefID” is a URL identification used to reference to its predecessor. The “CommonPrefix” is a number of common characters referenced to its predecessor. The “diffURL” is the tail of the URL that does not common to its predecessor. The “Lchild” and “Rchild” are the pointers to the node of left subtree and right subtree respectively.

| | | | | |
|---|---|---|---|---|
| RefID | CommonPrefix | diffURL | Lchild | Rchild |

**Figure 13** A node structure of the AVL tree

The first encountered URL, which is the root, is assigned with the “RefID” 0. The “RefID” is increment by one for every new encountered URL. Please note that the value in the field “RefID” of the root can be neglected. The common prefix is set to zero, and the full URL is stored. The next incoming URL must be compared with every node along the path between the root node and its predecessor to find the maximum common prefix. The reference ID of a new node is then point to that node, and the common prefix is set to the number of common characters, and the remaining of the URL is then stored. This is the way the compression is done.

Figure 14 shows an example of an AVL tree constructed from the four following URLs: "*http://www.sun.com/*", "*http://www.sgi.com/*", "*http://www.sun.com/news/*", and "*http://www.sun.com/news/archive/*". The first URL always become the root of the tree: this is "*http://www.sun.com/*". Its reference URL ID is set to be zero, its shared prefix is also set to be zero and the whole URL is stored in the third field. The next URL, "*http://www.sgi.com/*" is compared to "*http://www.sun.com/*". Since it is "smaller", the URL is then attached to the left subtree of "*http://www.sun.com/*". This URL has a 12 bytes common prefix of "*http://www.s*", so it is stored as "*gi.com/*" with a reference ID 0. The next URL, "*http://www.sun.com/news/*", is "greater" than "*http://www.sun.com/*", so a new node with "*news/*" is added as the right subtree "*http://www.sun.com/*" with 19 bytes common prefix. The last URL, "*http://www.sun.com/news/archive*", it compared with the root and is "greater" than both "*http://www.sun.com/*" and "*http://www.sun.com/news/*", so a node with "*archive/*" is added as the right subtree of "*news/*". The common prefix is 24 bytes long ("*http://www.sun.com/news/*") and the node is pointed back to URL ID 2.

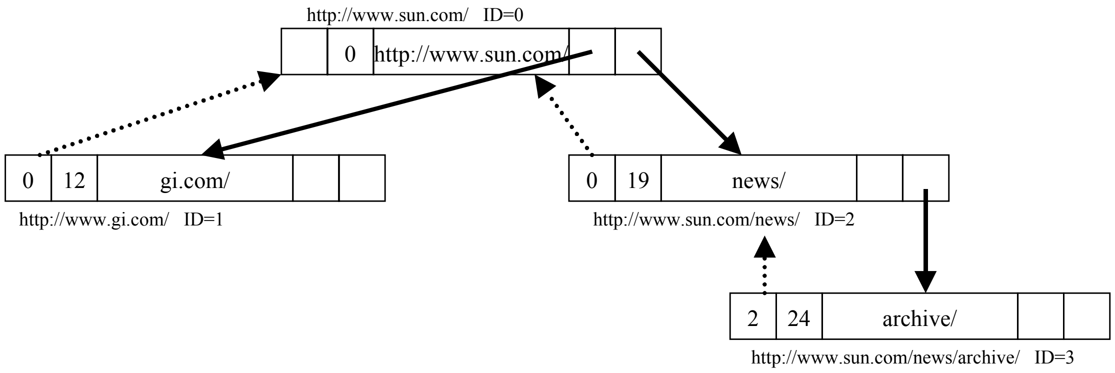


**Figure 14** Compressed URL on an AVL tree

Retrieving any URL from the tree is very simple. The full URL can be re-constructed by following the path and concatenate all URL from the field "diffURL". This approach can fulfil all the requirements of search engines and web spiders. The URLs can be added, searched, and the full URL can be retrieved at any time.

In KSpider, the AVL tree is implemented by three arrays. The first array, the "TreeNode", contains a list of nodes of the AVL tree as shown in Figure 15. It contains a Left node ID, right node ID, and the height. The height is split into two nibbles, because the compiler (g++) will use 96 bits or 12 bytes instead of 8 bytes to hold the structure in order to align the data in word alignment. The variable length data were stored on the second array called "CompressedURL", as shown in Figure 16. The "CompressedURL" is accessed through the third array, the "DataPtr" as shown in Figure 17. These tree arrays must be pre-allocated. The reason behind this implementation is because the compiler can produce an optimum code if the data can be aligned in $2^n$ bytes boundary. Furthermore, the "TreeNode" is used only for adding a new URL and can be discarded if all URL is already processed. On another word, after the tree has been completely constructed, the "TreeNode" is no more required.

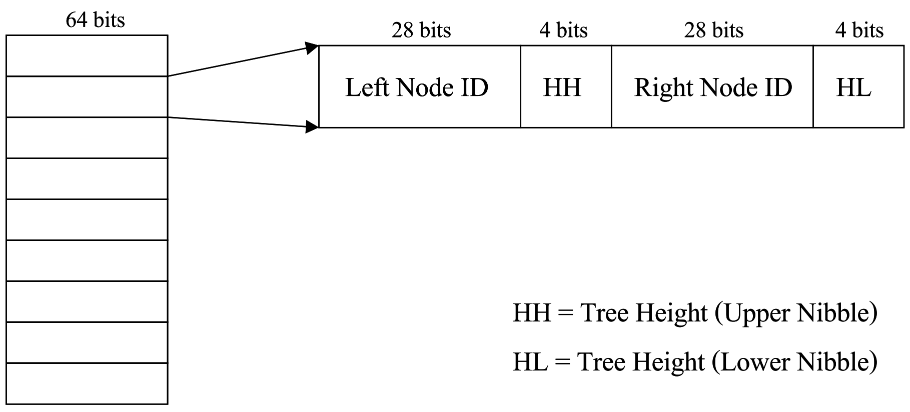

64 bits
28 bits
4 bits
28 bits
4 bits
Left Node ID
HH
Right Node ID
HL
HH = Tree Height (Upper Nibble)
HL = Tree Height (Lower Nibble)

**Figure 15** TreeNode

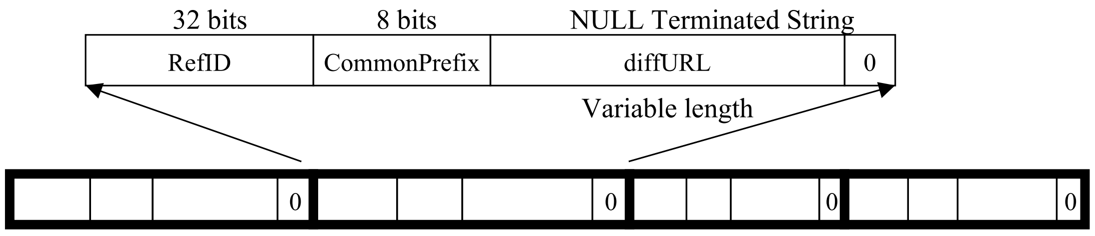


**Figure 16** CompressedURL

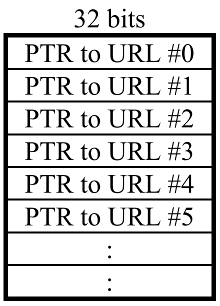


**Figure 17** DataPtr

The height of each node is split into two nibble and store on the 64-bit structure on the first array. This can reduce memory space, and the maximum URLs is equal to $2^{28}$ entries, or around 268 million URLs. Like in Figure 18, "TreeNode" and "DataPtr" can be viewed as a single array, with the ability to discard the first array when needed. A node on this tree can be accessed directly using “DataPtr”. This means that the tree can be accessed by using an index (through the “DataPtr”) or by searching the URL (through “TreeNode”) like a binary search tree.

The overhead for storing a single URL is equal to 18 bytes (8 bytes for a couple of left and right node ID in “TreeNode”, 6 bytes for the “CompressedURL” excluding the “diffURL”, and 4 bytes for the “DataPtr”). As explained above, the overhead can be reduced to 10 bytes when the “TreeNode” is discarded.

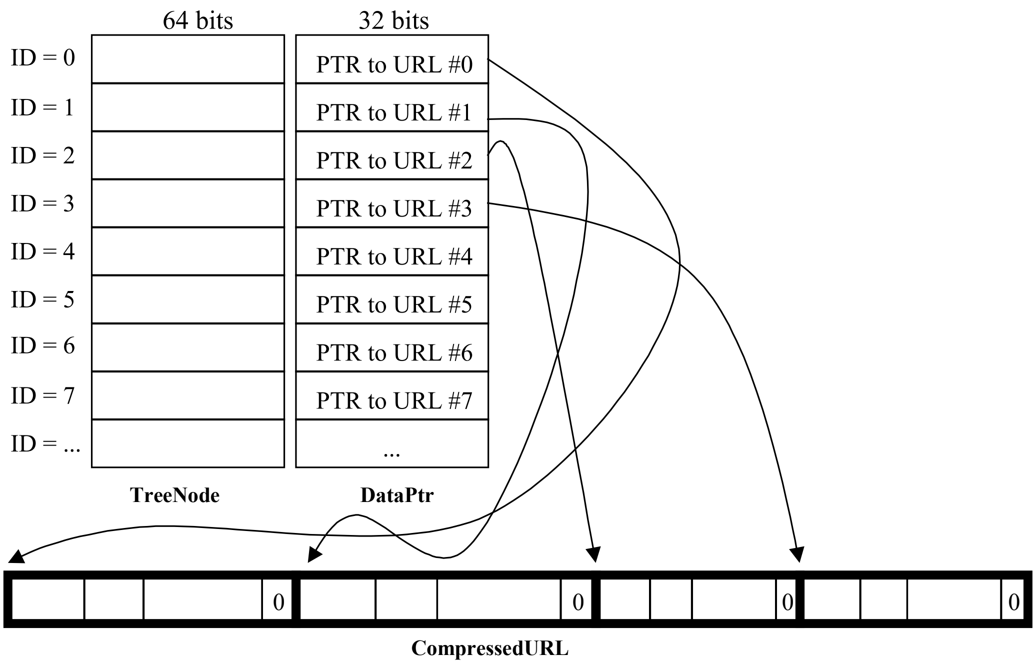


**Figure 18** Data Structure of an AVL Tree for URLs Compression

## 8. Parallel Storage

The data storage, as shown in Figure 19, requires many features for achievement of system efficiency as the following.

1. Efficient in the term of space.
2. Fast storing (write from many threads)
3. Fast retrieving (read from many threads)

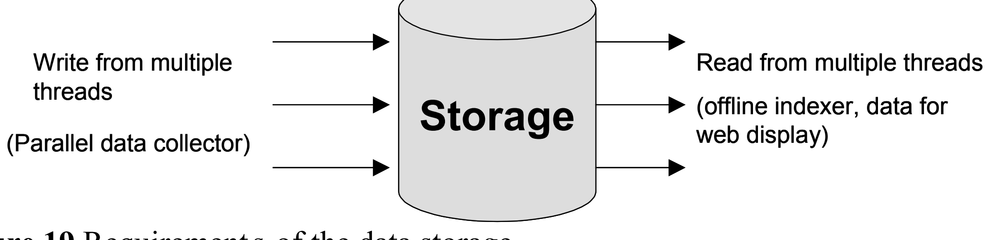

Write from multiple threads
(Parallel data collector)
Storage
Read from multiple threads
(offline indexer, data for web display)

**Figure 19** Requirements of the data storage

The data from the data collector will be passed to the Data Processor, and then will be buffered at the 'Incoming Data Pool' inside the Storage Manager (From Figure 20). The compressor thread will pull the data from the Incoming Data Pool to compress, and send the compressed data to the 'Compressed Data Pool'.

There is one 'Saver Thread' per one storage device, as shown in Figure 20. Note that any storage device should have only one partition in them. The 'Saver Thread' will try to compete to get the data from the 'Compressed Data Pool'. When the 'Saver Thread' got the data, it will then store the data on the physical storage. Any 'Saver Thread' that is busy writing data to disk can not compete to get the data from the 'Compressed Data Pool', only the thread that is not busy can. If there are many 'Saver Thread' waiting for a data, the Linux's pthread library will schedule theses thread in first-in-first-out manner.

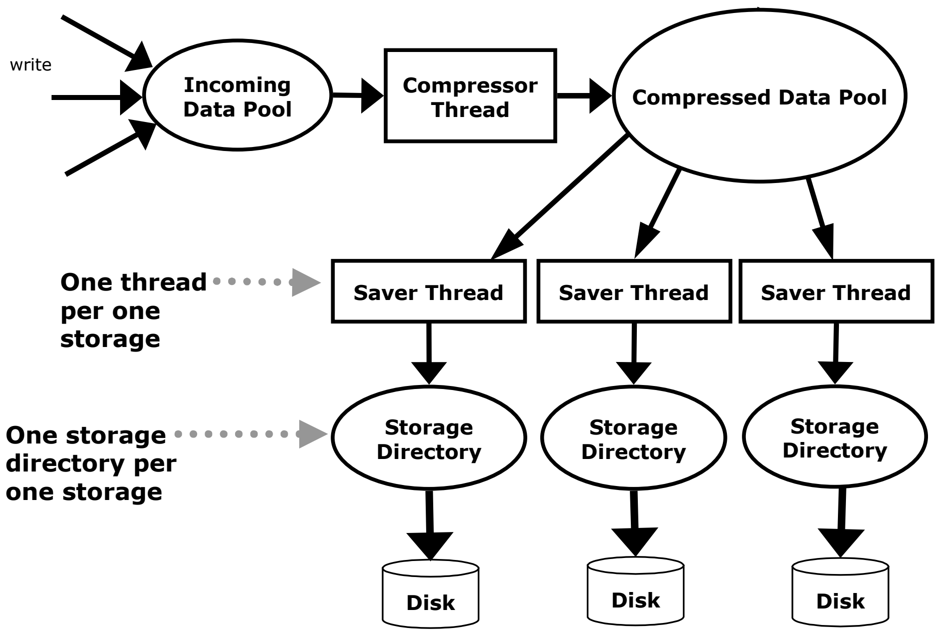


**Figure 20** Overall design of the data storage (Storage Manager)

To prevent internal fragmentation, the compressed data will be packed together on some large file. On each 'Storage Directory', it contains an index that contains the file IDs for which every compressed data are stored, an also contains offsets that point to the beginning of data in the file.

## 9. URL Extractor

The role of URL Extractor is to extract any links in the HTML pages. As shown in Figure 21, it consists of two modules, the Link Extractor and the URL Normalizer. The Link Extractor parses the HTML page, and sends any links it found to the URL Normalizer. The URL Normalizer converts any relative URL into an absolute URL. For example, the relative URL "*../images/face.jpg*" that is found in "*http://www.ku.ac.th/news/200301.html*" can be normalized to "*http://www.ku.ac.th/images/face.jpg*"

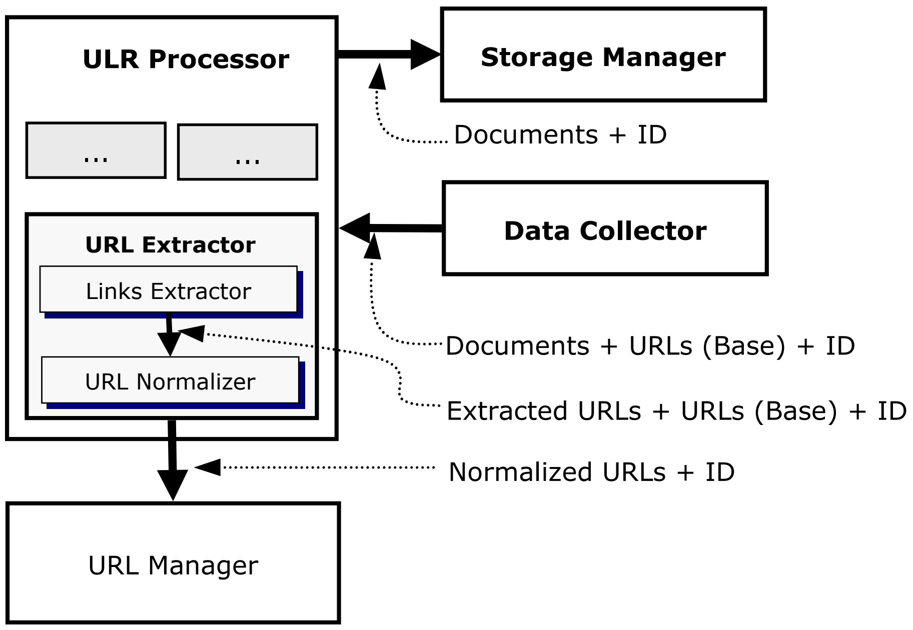


**Figure 21** Data flow diagram of the URL Extractor

The Link Extractor uses a finite state machine in parsing an HTML page. The state machine is designed to accept some common wrong syntax in the HTML.

The HTML files usually contain several HTML tags and some text body. Only fractions of HTML tags that indicate links to another HTML page or another resource are considered in the extraction process. These HTML tags (World Wide Web Consortium, 2003), or usually called 'element', are:

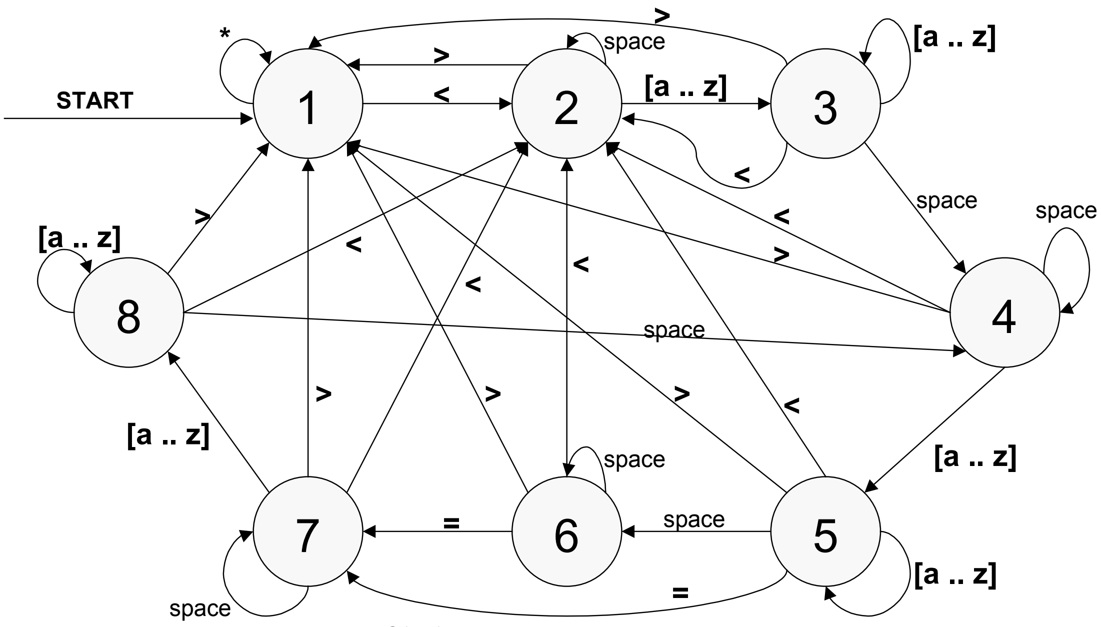
1. A
2. IMG
3. FRAME
4. BASE
5. AREA
6. LINK

Every element has its own set of attributes. For example, the 'A' element may have a 'href' attribute, but the 'IMG' element may have a 'src' attribute. Some vendors have added some attribute into some tag. For example, some web browser support a 'lowsrc' attribute, which is used to specify a lower resolution or lower size version of an image. The summary of HTML elements and attributes that KSpider recognizes are illustrated in Table 1. Note that the 'BASE' element is not actually a link, but it is mandatory for correctly normalize the URL.

**Table 1** HTML elements and attributes

| Element | Attribute |
|---|---|
| A | href |
| IMG | src, lowsrc, lowres |
| FRAME | src |
| BASE | href |
| AREA | href |
| LINK | href |

Every HTML tags starts with '<' and end with '>'. After '<' it may contains some white-space. After that, the element follows. The HTML tag may or may not have any attribute. A state machine in Figure 22 is used to recognize the HTML tags.



```
State Name
1 IN CONTENT
2 FIRST WITE SPACE
3 ELEMENT
4 SECOND WHITE SPACE
5 ATTRIBUTE
6 THIRD WITE SPACE
7 EQUAL SIGN
8 VALUE
```

**Figure 22** State Diagram

Every links extracted by the Link Extractor are then passed to the URL Normalizer. The role of the URL Normalizer is to normalize the URL. There are several kinds of URLs that need to be normalized, such as:

1. Relative URL, such as "/images/", which is needed to be concatenated with the base URL.
2. Relative Directory, such as '/../' or '/./'.

3. Escape code, such as '%FE%96'
4. Change the host name part to be lowercase letter.
5. The default port (80 for HTTP) is omitted

## Other design and implementation techniques

### 1. Robot Exclusion Standard

To improve performance, the robot exclusion standard file "*robots.txt*", are kept in the spider's cache. The cache itself is an open hash. Each hash entry contains the host name, and the ID of the robot exclusion standard file. From the experimental, most of the web servers in Thailand do not have this file; hence most of the entry in the cache contains only the host name.

The '*robots.txt*' file will be parsed and kept only some entries that has an effect to the spider, i.e., comments and any other section that belongs to the other spider are discarded.

### 2. Resume techniques

The data structure used in our spider is either primitive data type or a record or an array, so it is easy to use 'mmap' to map these variables from persistent file storage. The OS is responsible to keep the data in the virtual memory and on the disk consistent. The spider can periodically inform the OS to flush any updated memory region into disk.

When the user want to temporary stop the spider, the spider will tell the OS to flush all the 'mmap' data and then exit. Once the spider starts again, it will then 'mmap' the old memory data and continues its operation using the previous state from the persistent storage.

It the case of power failure or system crash, the spider can also resume its operation using the persistent file storage.

### 3. Avoiding spider traps

#### 3.1 Symbolic Links

The symbolic links trap can be detected by finding any repeated occurrence of directory name. A simple hash function with small hash table is used to check for a duplicate directory. The hash table and hash function is a linear probing hash. Each slot in this table keeps a reference to the directory name.

For every directory name, if a directory made a collision, it is then check if it is really duplicated with another directory name.

### 3.2 Excessive Page Size

The spider has to check for the page size if it is available, or it can stop downloading if the data is getting too large. Literature on web analysis (Sanguanpong et al, 2000) shown that the maximum size of web pages lies between 1 KBytes to 1 MBytes. Human operator can pre-configure to guide the spider to stop downloading if the page sizes exceed the predefined configuration.

### 3.3 CGI Generated Pages

CGI generated web pages are usually a result of query. They usually contains some signature of being a CGI generated page, such as the present of ".cgi" or the present of "?" in the URL. These URLs can be filtered out easily using regular expression matching.

Another kind of dynamic generated page are '.php' and '.asp', these pages are become very common. These pages may contain hyperlinks, which some of them lead to an infinite number of URLs. The session ID appended to the URL is one of them. There is no known solution to this problem yet. These pages need to be filtered out manually.

## 4. <u>Fine tuning Linux for performance maximization</u>

Linux is very customizable, both to the kernel's parameters and running environment parameters. Here are some parameters that can be fine-tuned to fit the spider's need.

### 4.1 Maximum stack size

The default stack size is 8 Mbytes. When creating a new thread, each thread will have its own stack, occupied 8 Mbytes of virtual memory. With hundreds of threads, the virtual memory address space can be exhausted easily. The stack size can be limited to a much lower limit, without recompiling any library, using the '`ulimit`' command line. In 'bash' shell, the command '`ulimit -s`' can be used to query the current stack size limit. The same command can also set the stack limit by specifying the desire stack size after the '-s' option.

Figure 23 shows an example of using ulimit to query the stack size, which were 8192 kilobytes. The stack limit is then set to 128 kilobytes.

```
$ ulimit -s
8192
$ ulimit -s 128
$ ulimit -s
128
$
```

**Figure 23** An example o f using ulimit to limit the stack size

The drawback of using a smaller stack is that the application can not use too much or too large local variable. A special care must be taken when writing a recursive application too, because each call to a subroutine requires a stack frame to store the return address, local variable, and parameters.

## 4.2 Local port range

Linux uses a local port number from a limited size pool when initiate a TCP connection. Despite the fact that modern Linux kernel and distributions set the default pool range to about 28000 ports, it may not be enough for a web spider. The local port range can be adjusted by the system administrator.

The current local port range can be obtained by getting the content of a file "*/proc/sys/net/ipv4/ip_local_port_range*" (technically, it is not actually a file). The pool can be set using the same file. Figure 24 shows an example of obtaining and setting the local port range. KSpider extends the local port range of 32768-61000 to 1024-65000.

```
# cat /proc/sys/net/ipv4/ip_local_port_range
32768   61000
# echo 1024 65000 > /proc/sys/net/ipv4/ip_local_port_range
# cat /proc/sys/net/ipv4/ip_local_port_range
1024    65000
#
```

**Figure 24** An example o f obtaining and setting the local port range

## 4.3 File system's option

The Linux's file system is a general-purpose file system. Some parameter can be fine-tuned to match the behavior of the spider, and hence gain more performance.

Nowadays, nearly all Linux distributions have adapted to use journalize file system called 'ext3'. The recovery time is very fast compared to 'ext2' file system, which once was the standard Linux file system. To create an ext3 file system, a program called 'mke2fs' is used. The '-j' option is needed to specify that the file system should be created with ext3 journal.

It is recommended to use one large partition per one disk to hold the spider's data for better performance. Other file systems, such as the root file system or a swap partitions, should reside in the other disks. The spider uses a small number of large files, therefore, a larger bytes-per-inode are better. The consequence is that the file system can holds less number of files, but the file system has more free space to contain data. The bytes-per-inode can be specified using the option '`-N`'.

The block-size is the unit size of the file system. Every file uses multiple of block-size bytes to store data on disk. A smaller block-size made the file system more efficient in term of space, meanwhile a larger block-size made the file system more efficient in term of speed. Like the bytes-per-inode, because of the way the spider store files, it is better to use the largest block-size file system (4096 bytes). The block-size can be specified using the option '`-b`'. Mke2fs also support specifying both bytes-per-inode and block-size using '`-T`'.

The reserved-blocks-percentage, while very useful to guard the system from an unexpected crash when some essential files can not be fully written on disk by a daemon or some administrative software, has no use for the spider's purpose. The default reserved blocks is 5 percent. The larger is the disk, the more space is waste. The reserved blocks can be adjusted higher, lower, or completely disable using '`-m`' option.

Figure 25 demonstrates how to create an ext3 file system with 4 Mbytes per inode, 4096 bytes block size, and zero percent reserved space. The option '`-i 4194304 -b 4096`' is equivalent to '`-T largefile4`'.

```
# mke2fs -i 4194304 -b 4096 -m 0 -j /dev/sda1
mke2fs 1.27 (8-Mar-2002)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
256 inodes, 262144 blocks
0 blocks (0.00%) reserved for the super user
First data block=0
8 block groups
32768 blocks per group, 32768 fragments per group
32 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 33 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
#
```

<u>**Figure 25**</u> An example o f creating an ext3 file system

Linux, by default, records the access time of every file in the ext3 file system. Such logging might be very useful for the system administrator for later analysis, but not for the spider. The logging of such information creates more overheads into the system. It can be simply turned off by passing '`noatime`' option to the kernel when mounting the file system. This option can be set up to automatically guide the kernel at the boot time by appending it to the forth field of the file called '*`/etc/fstab`*'. Figure 26 illustrates the portion of the '*`/etc/fstab`*' for enable this option.

```
.
.
/dev/sda1          /data1         ext3      defaults,noatime       1 2
.
.
```

**Figure 26** An example shows how to use the ‘noatime’ option

## RESULTS

Instead of downloading only HTML pages, images and other file types are also included to build database for another project. KSpider collected over ten million URLs under `.th` domain on about 24,000 servers. The collection took 7 days long without any interruptions. Downloading web pages from international link would impact the University's network performance because the campus network has a shared and limited bandwidth only 8 Mbps; and the link utilization is quite full all day long. Meanwhile, there is a lot of unused domestic link available inside the country. Under this circumstance, KSpider is, therefore, deployed to collect the data from the web servers, which is registered under `.th` domain. Please note that there may be some web servers whose domain are under `.th`, but they locate outside the country. However, only a small number of such servers exist (Aroonwattanamongkol, 2002) and their impact on international link utilization could be neglected.

### URL Compression, size/speed

The average size of URLs is about 59.5 bytes. The distribution graph of the length of the URLs is shown in Figure 27.

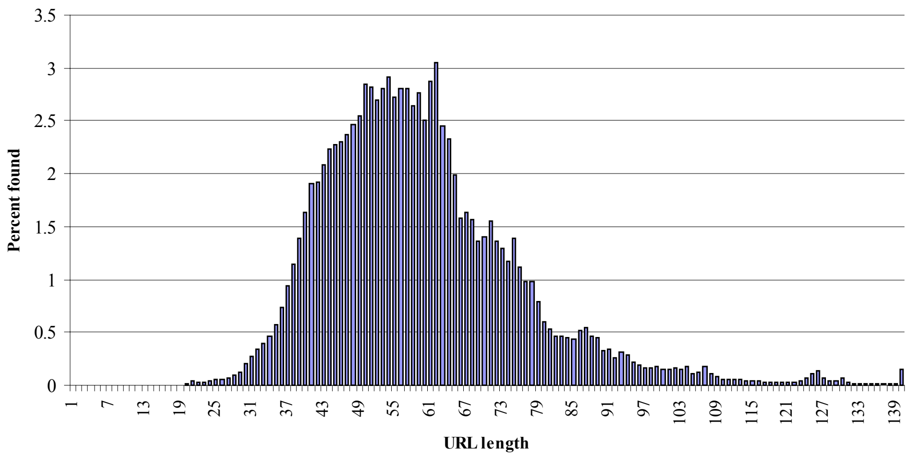


**Figure 27** URL length distribution graph (input)

The distribution graph of the compressed URLs is shown in Figure 28. Size of a compressed URL is about 26.5 bytes by averaged, including the entire overhead. The size would be reduced to 18.5 bytes when the "TreeNode" is discarded. So the actual compressed data size, excluding all overhead is just only 8.5 bytes. This yields about 55% of size reduction for web spiders' purpose with an ability to store and retrieve the URLs on the fly. Meanwhile, it yields about 69% of size reduction for search engines, where only the ability to retrieve the URLs is required.

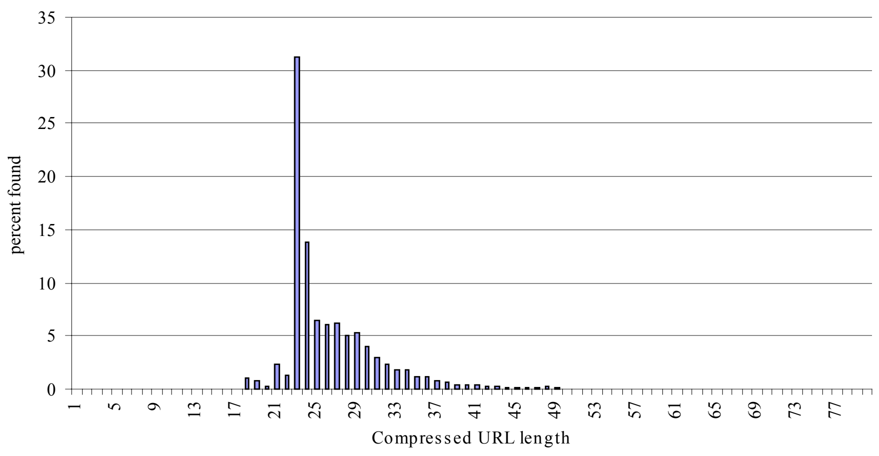


**Figure 28** Compressed URL length distribution graph

The access time is fast for both storing and retrieving. As illustrated in Figure 29 and Figure 30, it takes about 15 microseconds by average to compress and store a URL (Min = 1 µs, Max = 67 µs). Retrieving is much faster, as shown in Figure 33 and Figure 34. It takes only 1.4 microseconds by averaged to retrieve a full URL from any given URL's ID (Min = 1 µs, Max = 12 µs). Seeking any given URL is also fast, as shown in Figure 31 and Figure 32. It takes only 12 microseconds by average to search for any given URL (Min = 1 µs, Max = 53 µs).

For better visualization of these three values, the comparison graph between storing time, finding time, and retrieving time is shown in Figure 35 and Figure 36.

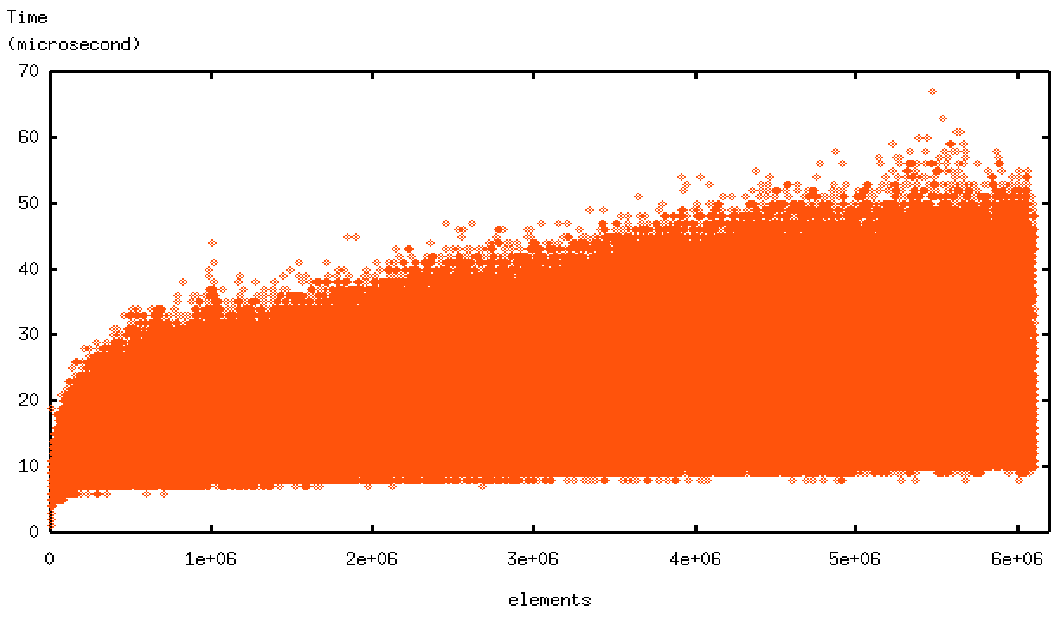


**Figure 29** Storing time (Linear scale)

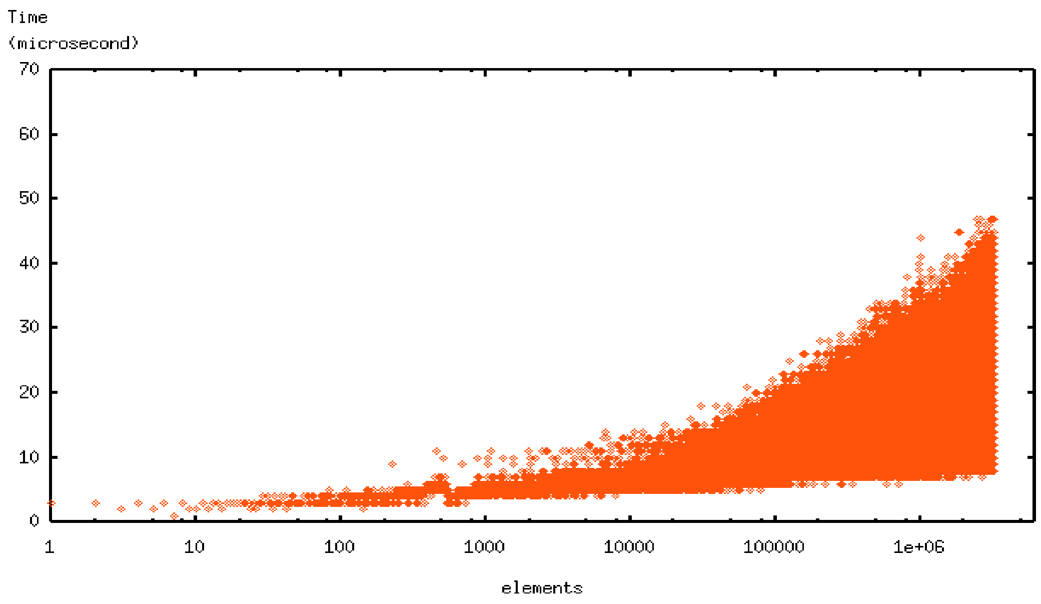


**Figure 30** Storing time (Log scale)

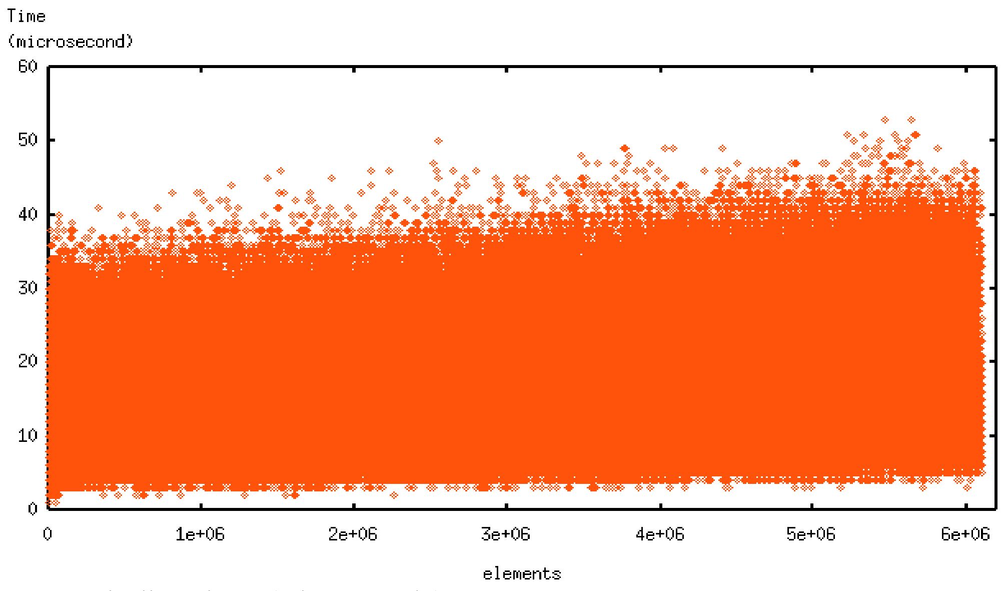


**Figure 31** Finding time (Linear scale)

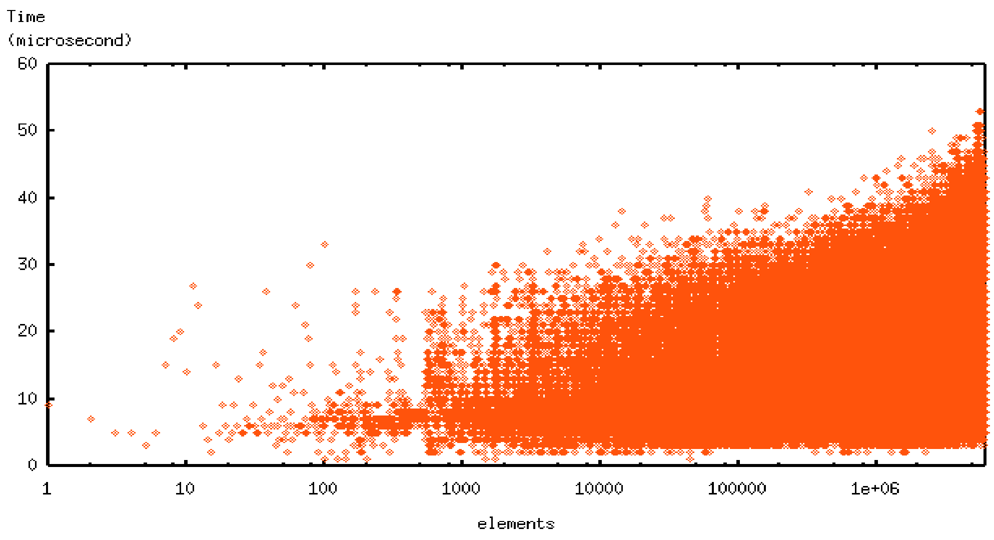


**Figure 32** Finding time (Log scale)

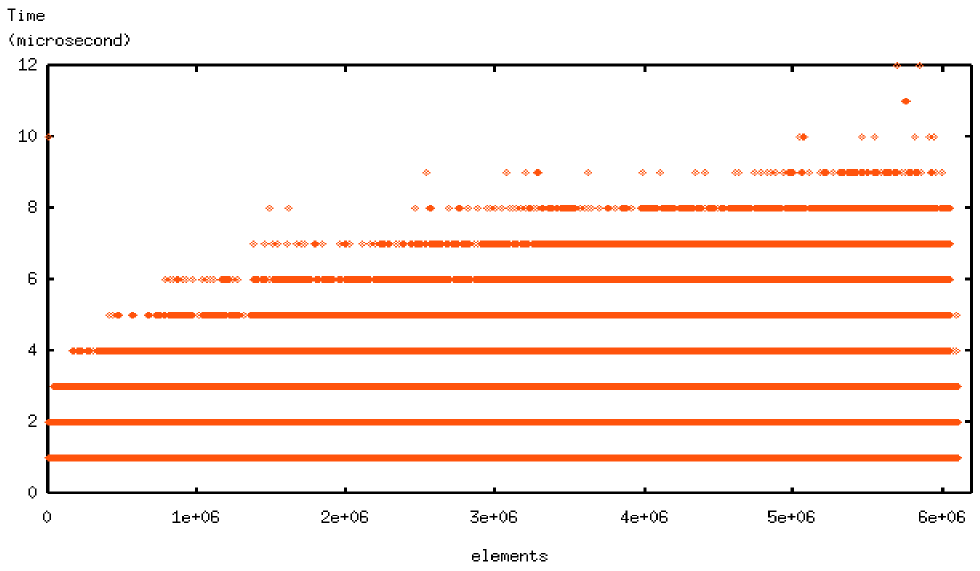


**Figure 33** Retrieving tim e (Linear scale)

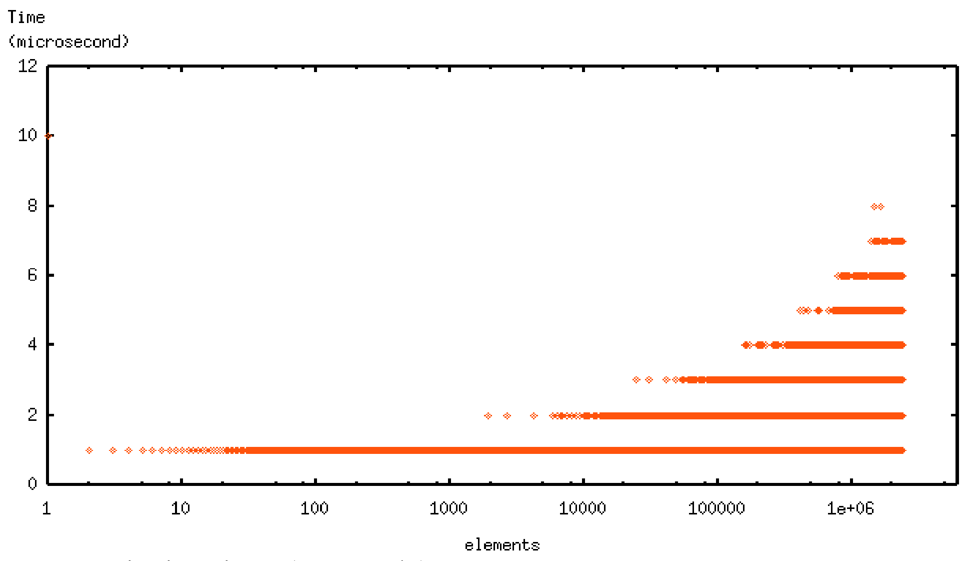


**Figure 34** Retrieving time (Log scale)

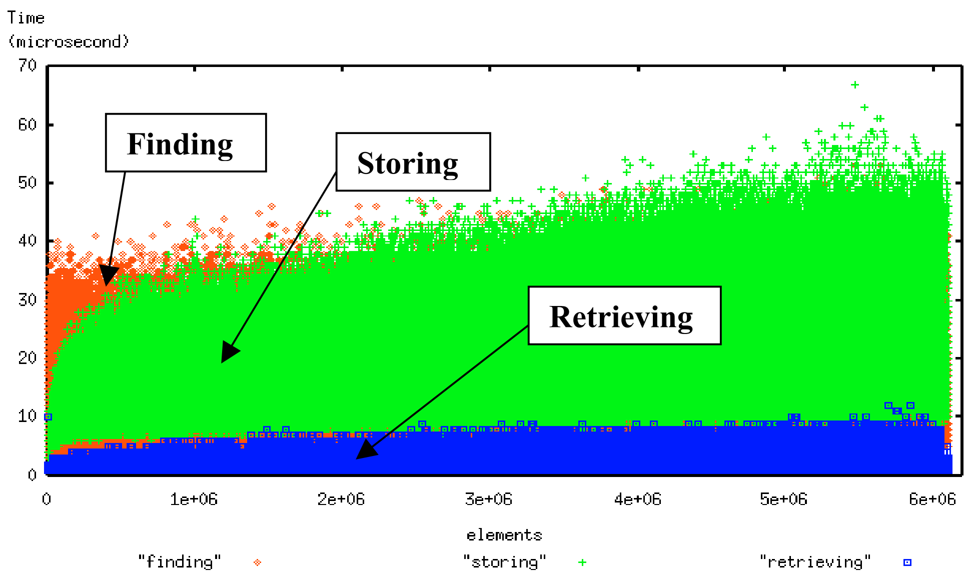


**Figure 35** Comparing between storing, finding and retrieving (Linear scale)

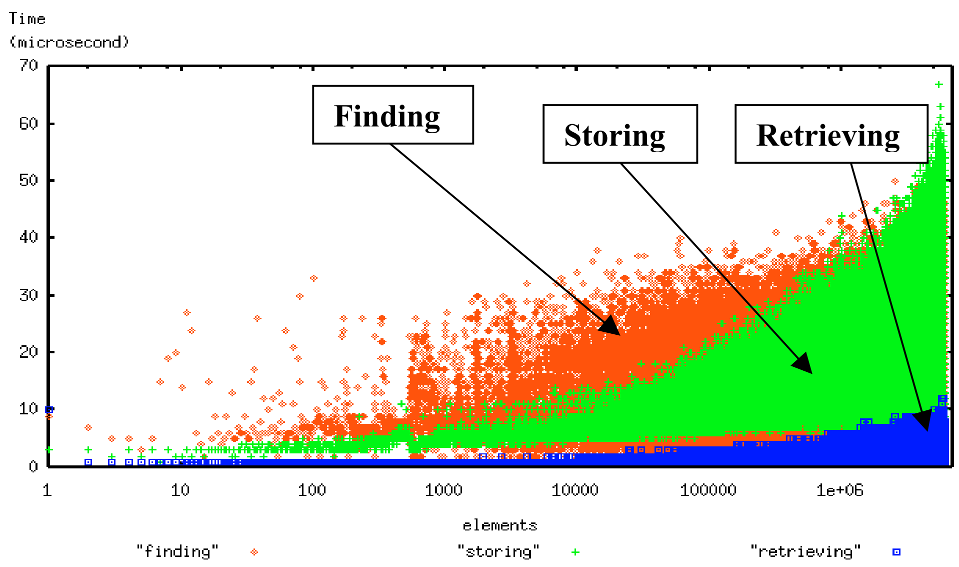


**Figure 36** Comparing between storing, finding and retrieving (Log scale)

## DNS Cache overhead/efficiency

The cache-hit rate of the DNS Resolver's cache is about 99.44 percent, which is very high. The 2048 buckets of 8 slots are prepared for the cache, which can hold a maximum of 16384 hosts. When cache hit, it uses about 1.6 microseconds by average to resolve a host name. The maximum resolve time takes about 23 microseconds, while the minimum time is 1 microseconds. The distribution graph of the resolving time, when the cache hits, is shown in Figure 37.

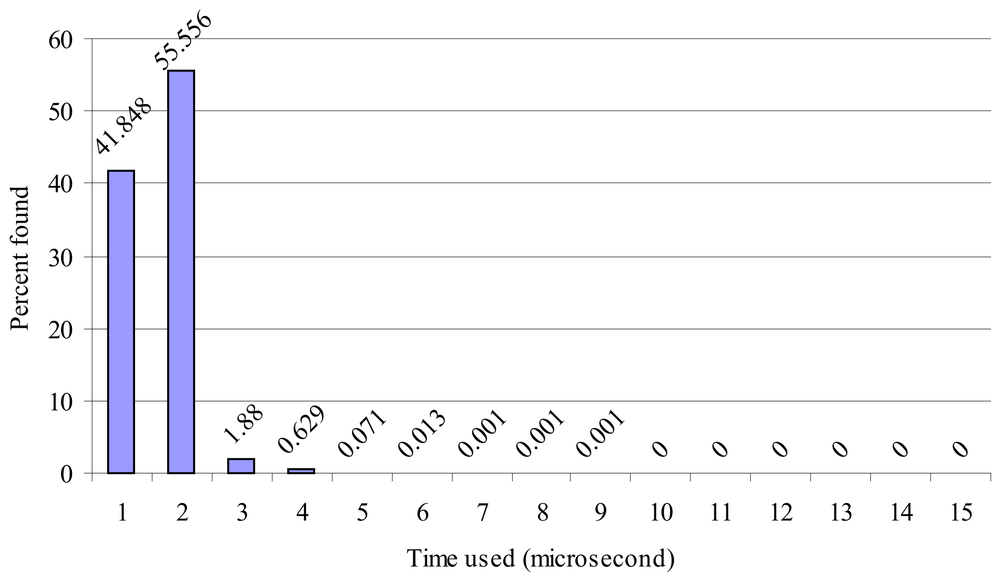


**Figure 37** Distribution of time used when the DNS cache hit

When the cache misses, the spider's resolver has to send a query to a DNS server. In this case, the respond requires resolving a name varies from 0.3 milliseconds to 10 seconds with an average time is 0.27 seconds. The operation time of the DNS cache when cache miss compared to the time used to do a query to the DNS server is negligible.

## Robot Exclusion Standard overhead/efficiency

The cache-hit rate of the Robot Exclusion Standard is very high, about 99.66 percent. This is because every found entry is kept in the cache without discarding. The number of entries in the cache is about 20,000 entries, which is very little compared to 6 millions of URLs. When the cache is hit, it uses about 3.24 microseconds by average to check any given URL. The maximum time is about 27 microseconds, while the minimum time is 2 microseconds. The distribution graph of the checking time when the cache hit is shown in Figure 38.

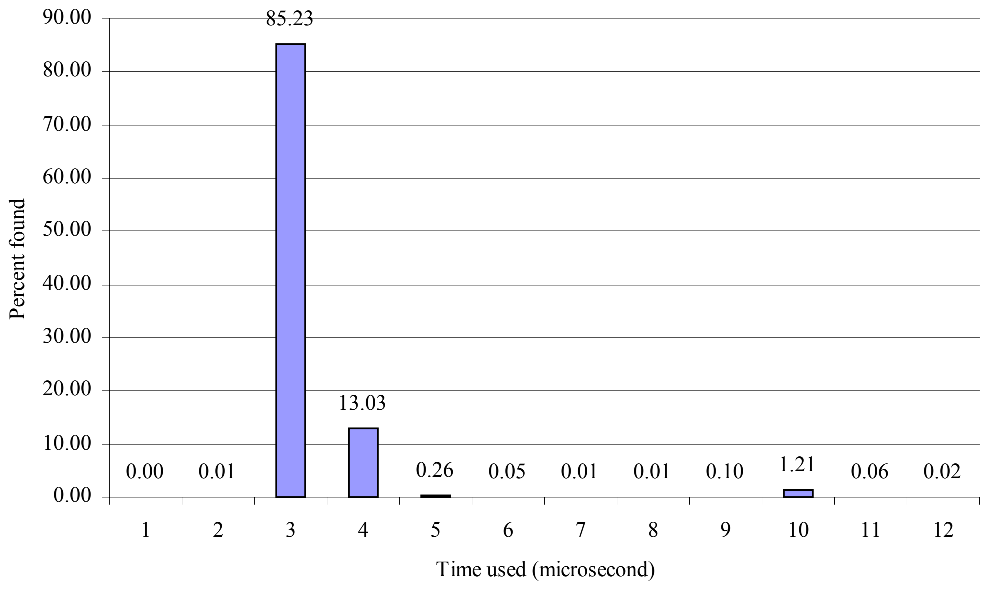


**Figure 38** Distribution o f time used when the robot exclusion standard cache hit.

When the cache is missed, a file 'robots.txt' must be fetched from the server. The fetching usually take much longer time than processing it, so the time used when caches miss is nearly equal to the time used to fetch the file.

The usage of 'robots.txt' under .th domain is very low. Only 1.03 percent of web servers implemented it.

## Work load distribution

The workload is distributed to the spiders by hash function. There are two hash functions that split the working set of the spider. The first hash function uses the full URL to decide which URL belongs to which spider server. From the test with 6 million URLs, with 2 to 10 servers, the hash function distributes the workload very well. Figure 39 show the URL distributions between the spider servers, for 2 to 10 servers.

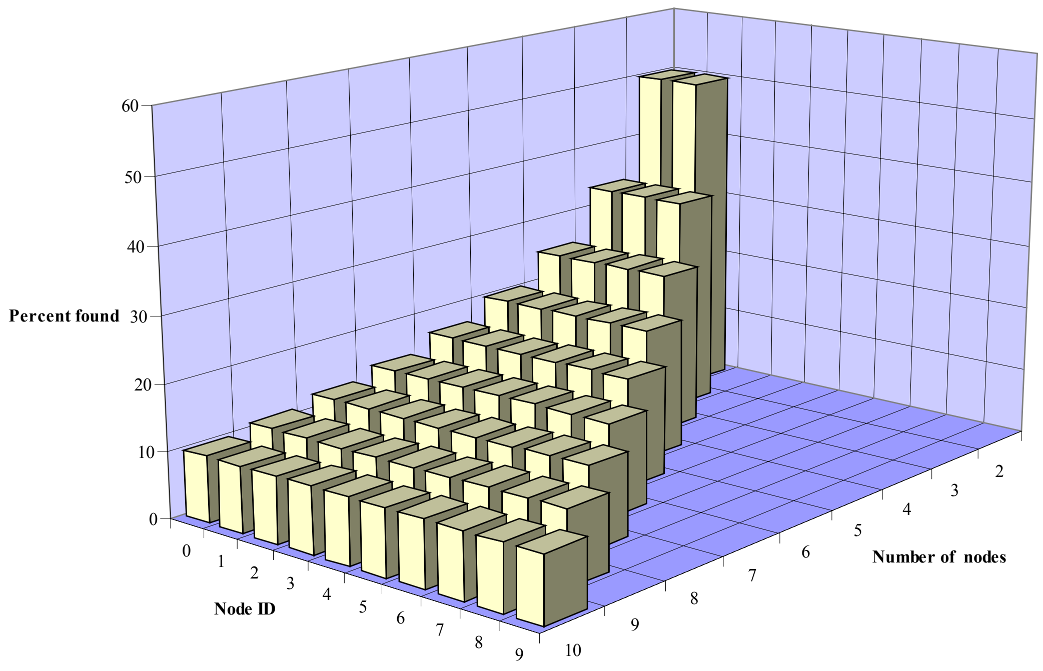


**Figure 39** Distribution o f URLs in 2 to 10 servers

The second hash function, as in equation (2), use the host name part of the URL to decide which URL belongs to which phase. The hash results are not as balanced as the first hash function, as shown in Figure 40, this is because there are some web servers that have much more URLs than the others.

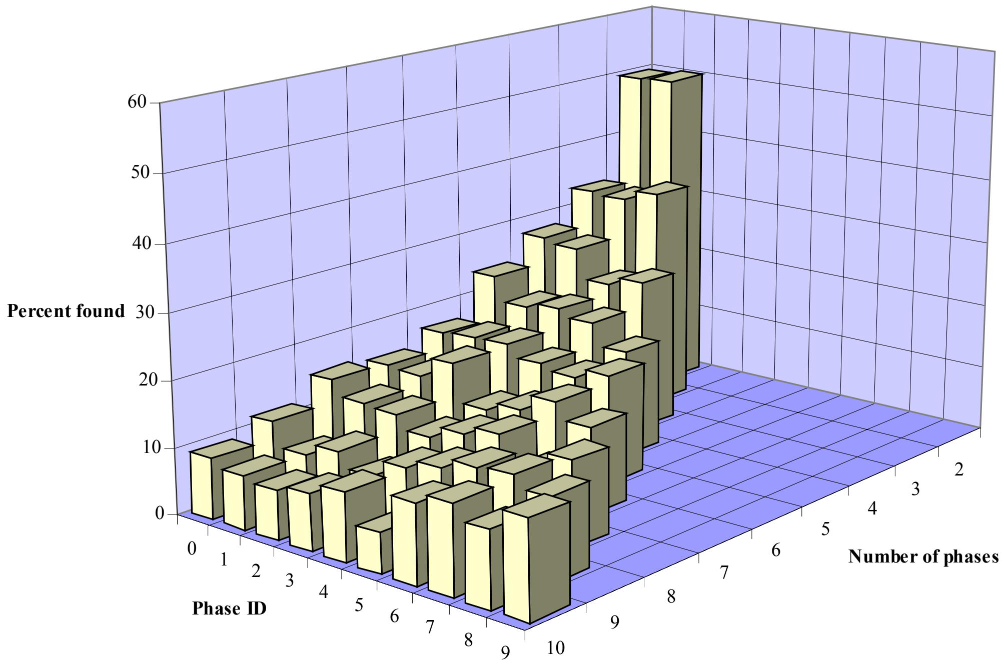


**Figure 40** Distribution of URL in 2 to 10 phases

## Memory usage

KSpider is designed to be able to handle at least 100 millions URLs by four machines. This means that a single machine is able to handle 25 millions URLs. The memory requirements are analyzed from two points of view. The first view is to analyze the virtual memory usage and the second one is to analyze the physical memory usage. Note that the analysis of memory requirements focuses on the large objects that requires a large amount of memory, or a large number of small objects that also require a large amount of memory. The details of any small number of small objects are not going to be discussed because they have very little effect on the performance.

### 1. Virtual memory

The virtual memory is used for the following proposes.

1. Store the binary image of the program itself, and any shared library that are needed. This part requires about 3 Mbytes of memory.
2. Store the URLs, requires about 27 bytes per URL. This part is a memory-mapped file. For 25 millions URLs per machine, this requires about 644 Mbytes of memory.
3. Store the state of the URLs used by the URL Manager to keep track of the URLs. It requires 1 byte per a URL. This part is a memory-mapped file.

For 25 millions URLs per machine, this requires about 24 Mbytes of memory.

4. Store the buckets of the Scheduler, requires about 1250 bytes per one element, or about 34 Mbytes of memory for 3600 set of host name and 8 entry for each set.
5. Store the data that is just fetched from the web server. It requires about 24 Kbytes per object. There are at most 300 objects (due to maximum 300 threads per machine) which the Data Collector must hold; this requires about 7 Mbytes of memory.
6. Store the thread's stack. The stack is limited to 128 Kbytes, so every thread uses 128 Kbytes for its stack. For 300 threads, about 38 Mbytes is required.
7. Store the index of the web pages, used by the Storage Manager. It requires 12 bytes per URL. For 25 million URLs, it requires about 286 Mbytes. This part is a memory-mapped file.

The virtual memory usage is summarized in Table 2.

**Table 2** Virtual memory usage

| **Purpose** | **Size (MBytes)** |
|---|---|
| Binary Image | 3 |
| URL | 644 |
| URL State (Scheduler) | 24 |
| URL Bucket (Scheduler) | 34 |
| Data (Fetched pages) | 7 |
| Data Index (Storage Manager) | 286 |
| Stack | 38 |
| Sum | 1036 |

## 2. Physical memory

From the virtual memory point of view (see section 1), not all of the virtual memory address spaces are kept in physical memory. Linux uses 'on demand paging' and 'copy on write' mechanism, which mean that any allocated virtual address space but never access will not actually use a physical memory. The physical memory requirements depend on the working set (the address space that the spider touch) of the spider.

In section 1, the top most memory consumer is the URL Manager, which uses the memory to store the URLs, and to store the state of the URLs. The URLs are accessed heavily throughout the running time of the spider, so all of these address spaces will remain in the physical memory. Anyway, the state of the URLs is not frequently accessed because the scheduler, as described in page 6, will access only a subset of the URLs. This is because the scheduler uses a sliding window. The window size in KSpider is 250,000 entries. This means the working set of the state of the URLs is about 250,000 bytes.

The working set of the Storage Manager, which requires 286 Mbytes virtual address space to keep the data index, is also required less physical memory than the virtual memory. The working set of the Storage Manager is also closed to 250,000 entries. This means the working set of the Storage Manager consumes about 3 Mbytes. Figure 41 shows the estimated memory requirement of the web spider.

The estimated working set size (estimated physical memory usage) is shown in Table 3. A comparison between the physical and virtual memory usage is shown in Figure 41.

**Table 3** Estimated working set size (physical memory usage)

| Purpose | Size (MBytes) |
|---|---|
| Binary Image | 3 |
| URL | 644 |
| URL State (Scheduler) | 0.25 |
| URL Bucket (Scheduler) | 34 |
| Data (Fetched pages) | 7 |
| Data Index (Storage Manager) | 3 |
| Stack | 38 |
| Sum | 729.25 |

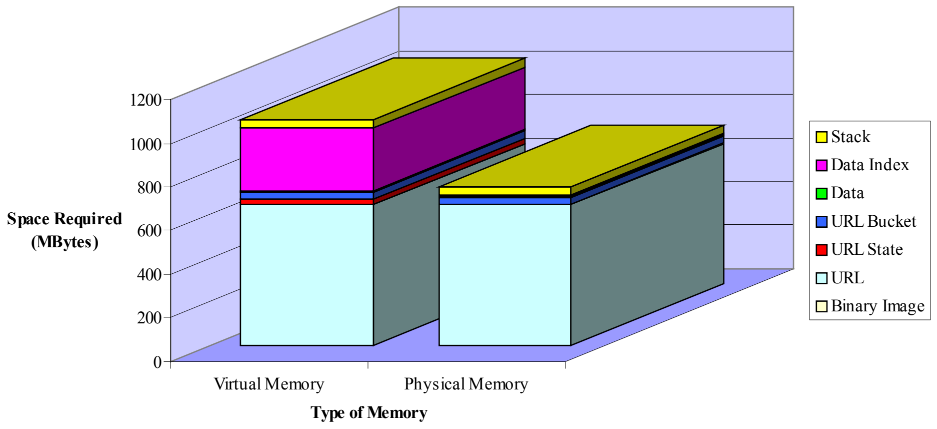


**Figure 41** Estimated memory requirement of the web spider

## Descriptor Usage

A descriptor is used in many modules. Some descriptors are temporary used such as a descriptor for reading a configuration file. Some descriptors are used throughout the lifetime of the spider. Some descriptors are used throughout the lifetime of the spider. Such descriptors are summarized in Table 4.

**Table 4** Summary of descriptors usage

| Module Name | Descriptor used | Purpose |
|---|---|---|
| DNS Resolver | 1 | UDP socket for contacting DNS server |
| Communicator | 1 | UDP socket for exchanging URLs |
| Data Collector | Number of threads | TCP socket for connecting to web servers |
| Storage Manager | Number of disk partitions | File descriptor for writing compressed web data |

## Collecting speed

The collecting speed is measured separately from the other tests, because it involves in repeatedly fetching the same set of URLs for many times. A smaller set of URLs, about 400,000 URLs under ".th" domain is used. The measurement is running eight times, varies from one, two, three, and four machines, and with 50 threads and 300 threads each. The results are shown in Figure 42.

The gathering speed increased when more threads or more machines is used. There are some factors that limit the gathering speed in the test environment. One factor is that the spider's CPU utilization is easily saturated because every thread has to manage a fast stream of web pages. Incoming stream of data has to be parsed and compressed which both required high CPU resources. Actually, the CPU utilization already reaches 100% when the number of threads is closed to 300. KSpider achieves an average download rate of 618 URLs/sec with sustained 6 MBytes/sec with four machines under the campus environment mentioned before. This result yields an estimate speed of 53 million pages per day under Thai Internet infrastructure.

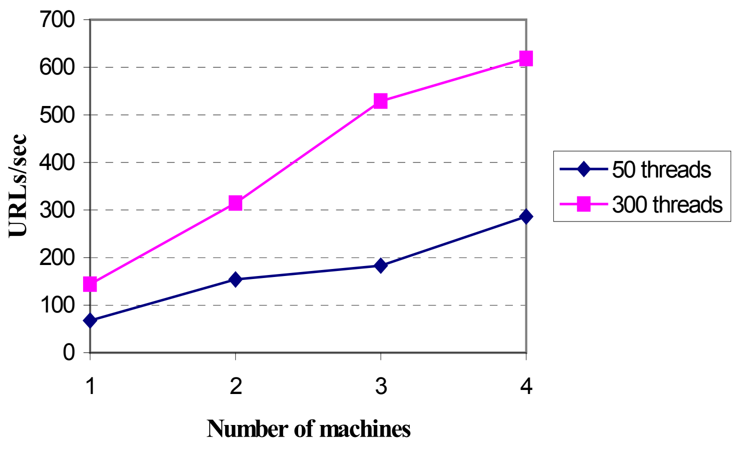


**Figure 42** Collecting speed

## Crawling Results

In this section, selected statistics related to web servers in Thailand are presented. More details about the statistics from the latest crawling are available at http://anres.cpe.ku.ac.th/.

A web server, in this thesis, is referred to as a web site that provides HTTP services. It is counted by a unique hostname, not a physical machine. Since a machine can support multiple web servers (with different hostnames) simultaneously using virtual host features. To precisely get the number of machines that are used as web servers, the IP addresses of machines are counted as well.

The KSpider was configured to download data from web servers whose names are registered under .th domain or their IP addresses are in Thailand. The list of all Thai IP addresses to be examined is created from the national exchange's route server (route-server.cat.net.th). Hence, crawling results include web servers under the .th domain and several other domains (.com, .net, .org, and etc.). The crawling was performed over a period of 7 days under only single machine to avoid overloading the campus network.

### 3. Size of Thai Web

Table 5 shows the summary of crawling results of 8,783,609 downloaded URLs, which consume over 155 gigabytes of disk space (compressed). Total 3,961,227 HTML documents on 24,124 web servers are found. The total size of HTML documents is around 54 gigabytes. For physical machines, there are 9,167 machines; therefore, in average, each machine supports 2.63 web servers.

**Table 5** Summary of collected results

| Type | Quantity |
|---|---:|
| Total URLs collected | 8,783,609 |
| Total size of data collected (GB) | 218 |
| Number of HTML documents | 3,961,227 |
| Total size of HTML (GB) | 54 |
| Number of web servers | 24,124 |
| Number of machines | 9,167 |

## 4. HTTP returned code

Table 6 shows the HTTP returned code. The 200 (OK) means the successful request of unique pages.

**Table 6** HTTP errors under 8,783,609 total requests

| Type | Quantity | Percent |
|---|---:|---:|
| 200 (OK) | 8,088,644 | 92.09 |
| 404 (Not found) | 545,615 | 6.21 |
| 401 (Unauthorized) | 75,827 | 0.86 |
| 403 (Forbidden) | 27,886 | 0.32 |
| 302 (Move temporary) | 19,437 | 0.22 |
| 301 (Move permanently) | 17,045 | 0.19 |
| 406 (Not Acceptable) | 4,116 | 0.05 |
| 503 (Service Unavailable) | 2,052 | 0.02 |
| 500 (Internal error) | 1,482 | 0.02 |
| 502 (Bad Gateway) | 547 | 0.01 |
| Others | 958 | 0.01 |

## 5. Percentage of server types (IIS, Apache, etc)

Web sites in Thailand are relied on Apache and IIS technology as shown in Table 7. The percentage of Apache and IIS are corresponded to the survey reported by Netcraft (http://www.netcraft.com/survey/)

**Table 7** HTTP server distribution

| Type | Quantity | Percent |
|---|---:|---:|
| Apache | 13,569 | 56.25 |
| Internet Information Server (Microsoft-IIS) | 7,787 | 32.28 |
| unknown | 1,604 | 6.65 |
| dozyGROUP WebServer | 227 | 0.94 |
| Netscape-Enterprise | 222 | 0.92 |
| TWH | 116 | 0.48 |
| Rapidsite | 60 | 0.25 |
| Ipswitch-IMail | 50 | 0.21 |
| IBM_HTTP_Server | 50 | 0.21 |
| Lotus-Domino | 48 | 0.20 |
| OmniHTTPd | 26 | 0.11 |
| Netscape-FastTrack | 16 | 0.07 |
| others | 349 | 1.45 |

## 6. Servers and documents classification by domain name

Statistics about the HTML documents for each domain are shown in Table 8. Over 70.02% of all documents are in the academic (.ac.th and .edu) and commercial domain (.co.th and .com).

**Table 8** Servers and documents classification by the domain name

| Domain | Number of web servers | Number of machines | Number of documents |
|---|---:|---:|---:|
| ac.th | 2,979 | 1,973 | 1,251,598 |
| co.th | 6,159 | 2,664 | 325,197 |
| go.th | 839 | 382 | 368,275 |
| in.th | 748 | 336 | 40,655 |
| mi.th | 67 | 21 | 18,194 |
| net.th | 102 | 81 | 63,015 |
| or.th | 651 | 400 | 281,684 |
| com | 10,385 | 2,249 | 1,176,188 |
| edu | 561 | 109 | 23,418 |
| gov | 4 | 4 | 62 |
| net | 764 | 403 | 104,917 |
| org | 481 | 245 | 116,086 |
| others | 384 | 300 | 191,938 |
| Total | 24,124 | 9,167 | 3,961,227 |

## 7. <u>Number of servers and number of documents</u>

The total number of web servers and total number of HTML documents on each domain is ranking as illustrated in Table 9 and Table 10, respectively. In Table 10, most of web pages found under the .com domain are pages generated by web boards.

**<u>Table 9</u>** Top-Twenty domains ranked by number of web servers

| Rank | Domain | Number of servers |
|---|---|---|
| 1 | msticker.com | 1473 |
| 2 | th.edu | 471 |
| 3 | diaryhub.com | 276 |
| 4 | ku.ac.th | 250 |
| 5 | storythai.com | 233 |
| 6 | chula.ac.th | 178 |
| 7 | th.com | 163 |
| 8 | thaitechno.com | 155 |
| 9 | police.go.th | 153 |
| 10 | psu.ac.th | 116 |
| 11 | cmu.ac.th | 114 |
| 12 | laopdr.com | 107 |
| 13 | doae.go.th | 100 |
| 14 | tu.ac.th | 95 |
| 15 | velocall.com | 88 |
| 16 | thaigov.net | 86 |
| 17 | rit.ac.th | 82 |
| 18 | kku.ac.th | 80 |
| 19 | su.ac.th | 63 |
| 20 | mahidol.ac.th | 61 |

**Table 10** Top-Twenty domains ranked by number of documents

| Rank | Domain | Number of documents |
|---|---|---|
| 1 | mthai.com | 292,391 |
| 2 | ku.ac.th | 271,246 |
| 3 | pantip.com | 115,487 |
| 4 | loxinfo.co.th | 84,427 |
| 5 | chula.ac.th | 80,770 |
| 6 | psu.ac.th | 76,407 |
| 7 | cmu.ac.th | 70,480 |
| 8 | bot.or.th | 70,434 |
| 9 | buu.ac.th | 60,901 |
| 10 | nectec.or.th | 54,396 |
| 11 | kapook.com | 49,991 |
| 12 | kku.ac.th | 48,799 |
| 13 | kanchanapisek.or.th | 40,866 |
| 14 | thai.net | 33,932 |
| 15 | moph.go.th | 33,524 |
| 16 | kmitl.ac.th | 31,664 |
| 17 | rit.ac.th | 29,660 |
| 18 | au.ac.th | 27,068 |
| 19 | phayoune.org | 26,863 |
| 20 | aecasia.com | 25,795 |

## 8. Characteristics of URL Links

Over all characteristics of hyperlinks in Thai web are illustrated in Figure 43 (only pages which has HTML links no more than 30 links are considered). Around 67% of all pages contain 1-10 outgoing links. Around 13% of web pages have no outgoing links. Average number of HTML links per page is equal to 15.

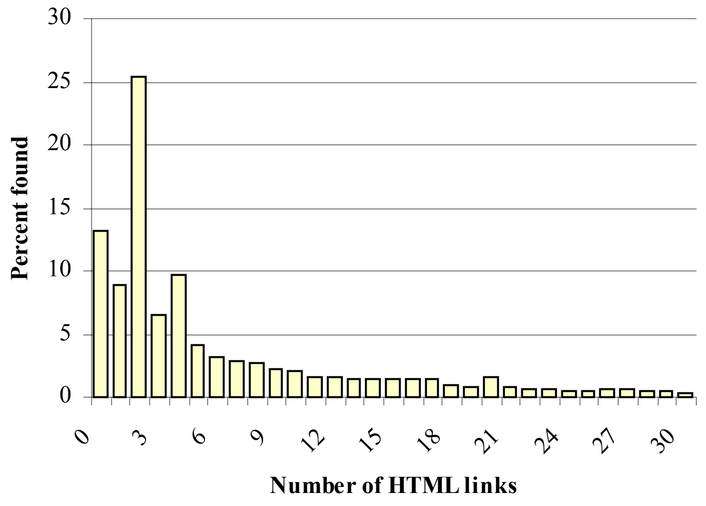


**Figure 43** Distribution of outgoing URL

## 9. Distribution of file extensions

File type classification has been done by the standard suffix used in file names e.g. .html, .htm, .jpg, .gif, etc. File names without suffixes are classified as unknown. From Table 11, 33.04% of all file extensions is HTML documents.

**Table 11** File extensio n classification

| Extension | Number of pages | % pages |
|---|---|---|
| .jpg | 2,474,971 | 28.18 |
| .gif | 2,279,698 | 25.95 |
| .html | 1,642,336 | 18.70 |
| .htm | 1,260,005 | 14.34 |
| unknown | 622,548 | 7.09 |
| .pdf | 128,877 | 1.47 |
| .asp | 102,478 | 1.17 |
| .php | 86,915 | 0.99 |
| .shtml | 71,862 | 0.82 |
| .doc | 42,252 | 0.48 |
| .xml | 27,461 | 0.31 |
| .png | 22,767 | 0.26 |
| .jpeg | 19,752 | 0.22 |
| .jsp | 1,687 | 0.02 |

## 10. Link connectivity

Over 60 million hyperlinks are found in the experiment. The quite rich connectivity property indicated by the number of links between each domain is shown in Table 12.

**Table 12** Links classifi cation by the second level domain name

| Domain | .ac.th | .co.th | .go.th | .in.th | .mi.th | .net.th | .or.th | others | Sum |
|---|---|---|---|---|---|---|---|---|---|
| .ac.th | 6,972,473 | 47,938 | 32,050 | 2,986 | 679 | 12,884 | 34,467 | 120,220 | 7,223,697 |
| .co.th | 2,652 | 6,276,278 | 4,871 | 226 | 58 | 433 | 5,052 | 359,254 | 6,648,824 |
| .go.th | 16,989 | 36,152 | 1,760,686 | 1,228 | 1,324 | 4,140 | 14,995 | 56,524 | 1,892,038 |
| .in.th | 1,085 | 12,381 | 1,388 | 271,555 | 24 | 69 | 1,095 | 6,541 | 294,138 |
| .mi.th | 568 | 1,003 | 1,049 | 33 | 50,003 | 62 | 497 | 1,462 | 54,677 |
| .net.th | 4,331 | 2,310 | 3,628 | 64 | 91 | 1,299,618 | 4,096 | 4,721 | 1,318,859 |
| .or.th | 7,807 | 11,010 | 122,140 | 222 | 199 | 2,828 | 3,363,298 | 43,333 | 3,550,837 |
| others | 47,622 | 204,916 | 38,428 | 4,853 | 1,158 | 7,237 | 56,874 | 39,561,025 | 39,922,113 |
| sum | 7,053,527 | 6,591,988 | 1,964,240 | 281,167 | 53,536 | 1,327,271 | 3,480,374 | 40,153,080 | |

## Future Works

There are many ways the web spider can be further improved. Detection to avoid infinite generated pages, the same pages under different host names (aliases), and mirror pages can greatly reduce waste bandwidth, disk spaces, and the processing time. The current bandwidth usage rely on a number of data collector threads and the delay time; a better bandwidth control mechanism to control the maximum bandwidth usage could ensure that the bandwidth used by the spider will not exceeded a predefine limit. The incremental update of web data could enable the spider to reduce the time and bandwidth required for update the web data.

# CONCLUSION

This thesis presents the design and implementation of the high performance cluster-based large-scale web spider. To handle a large number of URLs, it uses hash function to distribute URLs to multiple machines. On each machine, the URLs are compressed, and store in memory. The compression rate is about 55%, including all overhead with superior speed in both compression and decompression.

The workload is fully distributed for the spiders on the nodes in the cluster. Every web spider has identical functions, but they work on different data set. The communication between web spiders is minimized. There are only URL exchanges between them in asynchronous manner. The phase swapping techniques, while does not introduce any unnecessary synchronization to the system, can ensure that every web spider will not concurrently working on the same set of web servers in the same time. This made the spiders very scalable when adding more machines into the system.

The data storage is also efficient, both in term of space and time. Using one thread per one disk storage ensures that there will be only one thread writing to any single disk. The content is also written sequentially, so the disk bandwidth is maximized. Because the compression is used and the files are packed also made the storage very efficient in term of space. Statistics about Thai WWW are analyzed and presented from the crawling data.

In summary, this thesis has been carried out at the Applied Network Research Laboratory at Kasetsart University and has several contributions to the web spider research community in the engineering and practical aspects as the following:

1. Workload distribution techniques with communication minimization which yields high scalability

2. Phase swapping technique avoiding the overloading of remote web servers

3. In-memory URL compression with superior in both time and space utilization

4. Parallel DNS resolver scheme to increase DNS query performance and reduce DNS server workload with the enhance reliability.

5. Parallel data storage scheme to maximize disk speed capability

6. Very robust hyperlink extraction.

7. Efficient checkpoint technique for resuming the system after crashing or stopping.

# LITERATURE CITED

Aroonwattanamongkol, P., 2002. **Thailand Network Information Center (THNIC)**, Personal Communications.

Bharat, K., A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. 1998. The Connectivity Server: fast access to linkage information on the Web, In **Proc. of the 7th International World Wide Web Conference**, Brisbane, Australia.

Burner, M. 1997. Crawling towards Eternity: Building an archive of the World Wide Web, **Web Techniques Magazine**.

Cho, J., H. Garcia-Molina., 2000. The evolution of the web and implications for an incremental crawler. In **Proc. of 26th International Conference on Very large Data Bases**.

______, H. Garcia-Molina, L. Page, 1998. Efficient crawling through URL ordering. In **7th International World Wide Web Conference**.

Eichmann, D., 1994. The RBSE Spider- Balancing Effective Search Against Web Load, In **Proc. of the First International Conference on World Wide Web**, Geneva, Switzerland.

Fast Search & Transfer ASA, 2003, **AlltheWeb.com**, Available Source: http://www.alltheweb.com/

Fielding, R., Irvine UC, Gettys J., Compaq/W3C, Mogul, J., Compaq, Frystyk H., W3C/MIT, Masinter L., Xerox, Leach P., Microsoft, Berners-Lee T., 1999, **Hypertext Transfer Protocol -- HTTP/1.1**, Request for Comments: 2616.

Free Software Foundation, Inc., 2002. **GNU C Library**, Available Source: http://www.gnu.org/software/libc/ libc.html.

Google, 2003, **Google**, Available Source: http://www.google.com/

Heydon, A. and Najork, M. 1999. Mercator: A Scalable, Extensible Web Crawler. **World Wide Web**, Available Source: http://research.compaq.com/SRC/mercator/papers/www/paper.html

Koht-arsa, K., S. Sanguanpong, 2001. In-memory URL Compression**, National Computer Science and Engineering Conference**, Chiang Mai, Thailand.

Koster, M., 1993. **A Standard for Robot Exclusion**.

______, 1993. **Guidelines for Robot Writes**. Available Source: http://www.robotstxt.org/wc/guidelines.html

Michel, B. S., K. Nikoloudakis, P. Reiher, and L. Zhang. 2000. URL Forwarding and Compression in Adaptive Web Caching, In **Proc. of IEEE INFOCOMM 2000**, volume 2, Tel Aviv, Israel.

Mills, D.L, 1991. Internet time synchronization: the Network Time Protocol**. IEEE Trans. Communications COM-39**. Available Source: http://www.eecis.udel.edu/~mills/database/papers/trans.pdf

Najork, M., J. Wiener, 2001. Bread-first search crawling yields high-quality pages. In **10th International World Wide Web Conference**.

Nielsen, H. F., J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, C. Lilley, 1997. Network Performance Effects of HTTP/1.1, CSS1, and PNG, In **Proc. of ACM SIGCOMM'97**.

Oberhumer. M. F.X.J., 1996. **LZO data compression library**. Available Source: http://www.oberhumer.com/ opensource/lzo/

Pinkerton, B. 1994. Finding What People Want: Experience with the WebCrawler. In **Proc. of the Second International Conference on World Wide Web**.

Rabin, M. O., 1981. **Fingerprint by random Polynomials**, Report TR-15-81, Center for Research in Computing Technology, Harvard University.

Risvik, K.M., R. Michelsen. 2002. Search Engines and Web Dynamics**. Computer Networks**.

Sanguanpong, S., P. Pimsa-nga, S. Keretho, Y. Poovarawan, S. Warangrit, 2000. Measuring and Analysis of the Thai World Wide Web, In **Proc. of the Asia Pacific Advance Network**, Beijing.

______, S. Warangrit, and K. Koht-arsa, 2000. Facts about the Thai Web, **National Computer Science and Engineering Conference (NCSEC-2000)**, Bangkok.

Sedgewick R., P. Flajolet, 1995. **An Introduction to the Analysis of Algorithms**, Addison-Wesley Pub Co.

Sergey B., P. Lawrence, 1997. **The Anatomy of a Large-Scale Hypertextual Web Search Engine**.

Sterling, T., D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. E. Packer, 1995. Beowulf: A Parallel Workstation for Scientific Computation, In **Proc. of International Conference on Parallel Processing 95**.

University of California, 2000. **How Much Information?**, Available Source: http://www.sims.berkeley.edu/research/projects/how-much-info/internet.html

Wickermesinghe, R., R. Stata, J. Wiener, et al. 2000. **Link Compression in the Connectivity Server**, Technical Report, Compaq Systems Research Center.

World Wide Web Consortium, 2003. **Links in HTML documents**, Available Source: http://www.w3.org/TR/REC-html40/struct/links.html