HYPERDOG

Up To Date Web Monitoring Through Metacomputers

by
Jacob W. Green

HYPERDOG

Up To Date Web Monitoring Through Metacomputers

Jacob W. Green
jgreen@cnds.jhu.edu

Advisor: Dr. Yair Amir
yairamir@cnds.jhu.edu

A project report submitted to The Johns Hopkins University in conformity with the requirements for the degree of Master of Science in Engineering

Baltimore, Maryland

October, 2000

Acknowledgements

I would like to thank John Schultz for all his help. As a friend and colleague he helped me in countless ways whether he knew it or not. He is the co-creator of HyperDog and implementer of the LinkServers. I look forward to our continued adventures.

I am very grateful to my advisor Dr. Yair Amir for his faith in me and his support along this path. He has created a very precious thing at the Center for Networking and Distributed Systems that I am proud to be associated will and will always be grateful for.

I would like to thank all the people at the Center for Networking and Distributed Systems; John Schultz, Jonathan Stanton, Theo Schlossnagle, Cristina Nita-Rotaru, Claudiu Danilov, Ryan Borgstrom, and Dr. Baruch Awerbuch. They were a wealth of talent to bounce my crazy ideas off and I enjoyed the atmosphere and friendships we shared.

I would like to thank Dr. Mike Goodrich who helped expose me to the problems of current systems in the crawling and search communities. He was very helpful in bouncing ideas off while this work was being formulated.

I would like to thank my father Dr. Bernard Green for his love, support, and commitment to education. It was he who insisted I take a “computer” course when I got to college.

I would like to thank my mother Dr. Mary K. Green who passed away during the construction of this work. Her love, support, and perpetual interest in whatever I was doing will be missed.

I am grateful to Elyssa Back for her friendship and the summer that got me started on this area.

Finally I need to thank Jeffery LaSalle, my rock that reminds me of things outside of my education and career.

Jacob Green
October, 2000

My early work on metacomputers was supported by the Defense Advanced Research Projects Agency (DARPA) under the metacomputer grant F30602-96-0293.

Abstract

Search engines bring order to the content on the World Wide Web (WWW). They are the primary interface for Internet users to find information. They currently take 30 days or more to find the changes that occur on the WWW due to the enormous amount of data that must be crawled to find these changes. Yet the WWW is experiencing an explosive growth of content, which will either greatly increase the cost of maintaining search engines, or the rate that changes are found will progressively get worse.

HyperDog is a proposed solution to this problem. It uses a large Internet-wide metacomputer to track the changes on the WWW. This system filters out only the changes that occur and feeds them back to the search engines and other indexers. Because the percentage of new and changed pages per day is relatively small, HyperDog represents a significant bandwidth improvement. When search engines only have to manage the changes on the WWW, they can scale better with the growth of the WWW.

HyperDog represents a key application to take advantage of the new global metacomputers. By distributing the crawlers to many points in the network through these **metacomputers**, the crawling algorithm gains some advantages. The crawlers each scan a small part of the WWW, perform the necessary processing to determine if the pages have changed, and only returns back the small portion of the WWW that is new or has changed. Together the crawlers perform the majority of work and provide the search engines with a much smaller condensed form of the information they need.

This paper also examines the properties of these metacomputers in an attempt define these properties and encourage the development of other key constructive applications.

Contents

List of Figures

1 Introduction

The World Wide Web (WWW) is a large collection of information that is published and maintained by millions of individuals and organizations. The data is stored all throughout the Internet, both geographically and network topology wise. Millions of Internet users need to gain access to this information. The organization of this immense data set is therefore crucial to providing access to these millions of users.

The easiest and fastest way to access information on the WWW for most users is through the utilization of search engines. "You can look at the Internet as a large data structure. What is needed is a way to do quick searches on this data structure."[1] Search engines provide the primary search interface to this large data structure. However, the problem with relying on search engines for effective data retrieval is that any additions, deletes, or changes in the WWW data structure are not reflected in those search engines for 30 days. Therefore, although a search of the WWW can be completed in seconds, the time it takes for changes in the data structure to propagate to the search engines and be made available to the users is very large.

The root cause of this "out-of-date" problem is the algorithm used to discover and propagate changes in the WWW to the search engines. This process is called **web crawling**. Web crawling in its most basic form starts at one page on the WWW and follows the hyperlinks until it visits all of the pages. This depth-first search approach must be continuously run to monitor the WWW for changes. The WWW however is very large and distributed across a vast network, that is growing at an exponential rate of ~100% [BBGC00] a year.

Current web crawling approaches crawl from a logically centralized location. Through one access point on the network they try to contact all the other points on the network. While this approach works, it is severely hampered. The primary problem with this approach is the bandwidth limitations to the central location. Current crawlers visit and download a page every 30 days or more. At an average page size of 12K[2] [BBGC00], and static WWW size of 1 billion indexable pages [Sew], a crawler must download 12.3 Terabytes per complete scan. To complete a scan of all the pages on the WWW in 30 days would take a 37.9Mbit/sec continuous bandwidth stream from the Internet. To increase this scan rate to 1 day would require a 1137Mbit/sec continuous bandwidth stream. This problem only gets worse as the number of indexable pages increases by 100% a year. Figures 1, 2, & 3 demonstrate the effects this growth rate and increased scan rate has on the required bandwidth.

[1] Dr. Baruch Awerbuch, http://cs.jhu.edu/~baruch/

[2] Mercador project [AHMN99] reports the average HTML page size is 14K.

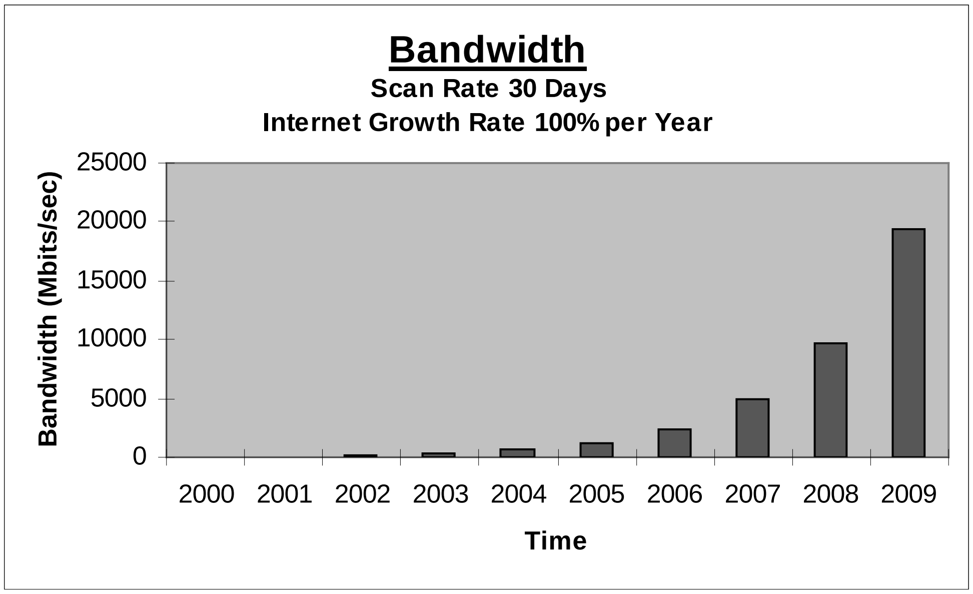


Figure 1: Bandwidth projected over time at a steady scan rate

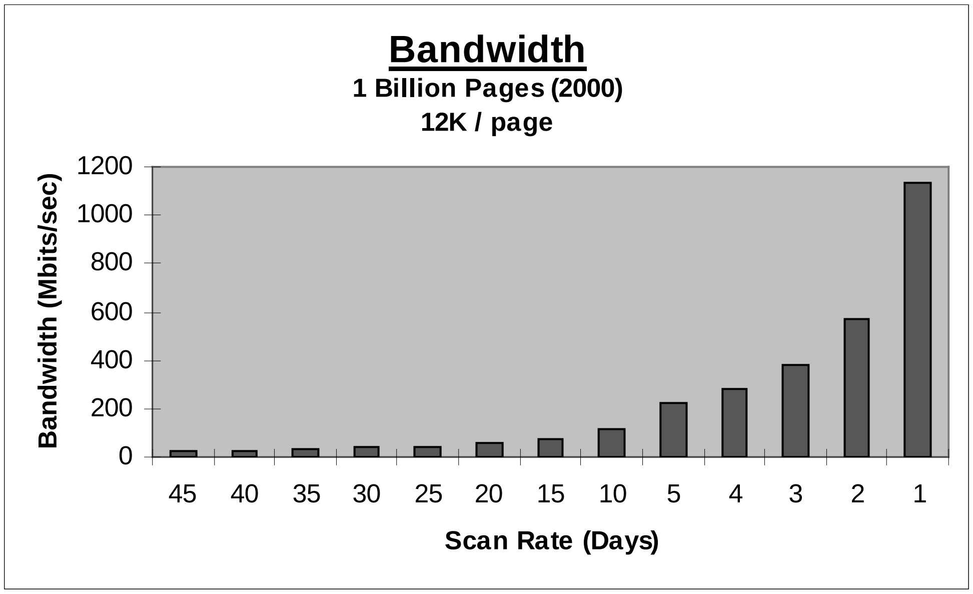


Figure 2: Bandwidth projected by varying scan rate as of today's WWW size

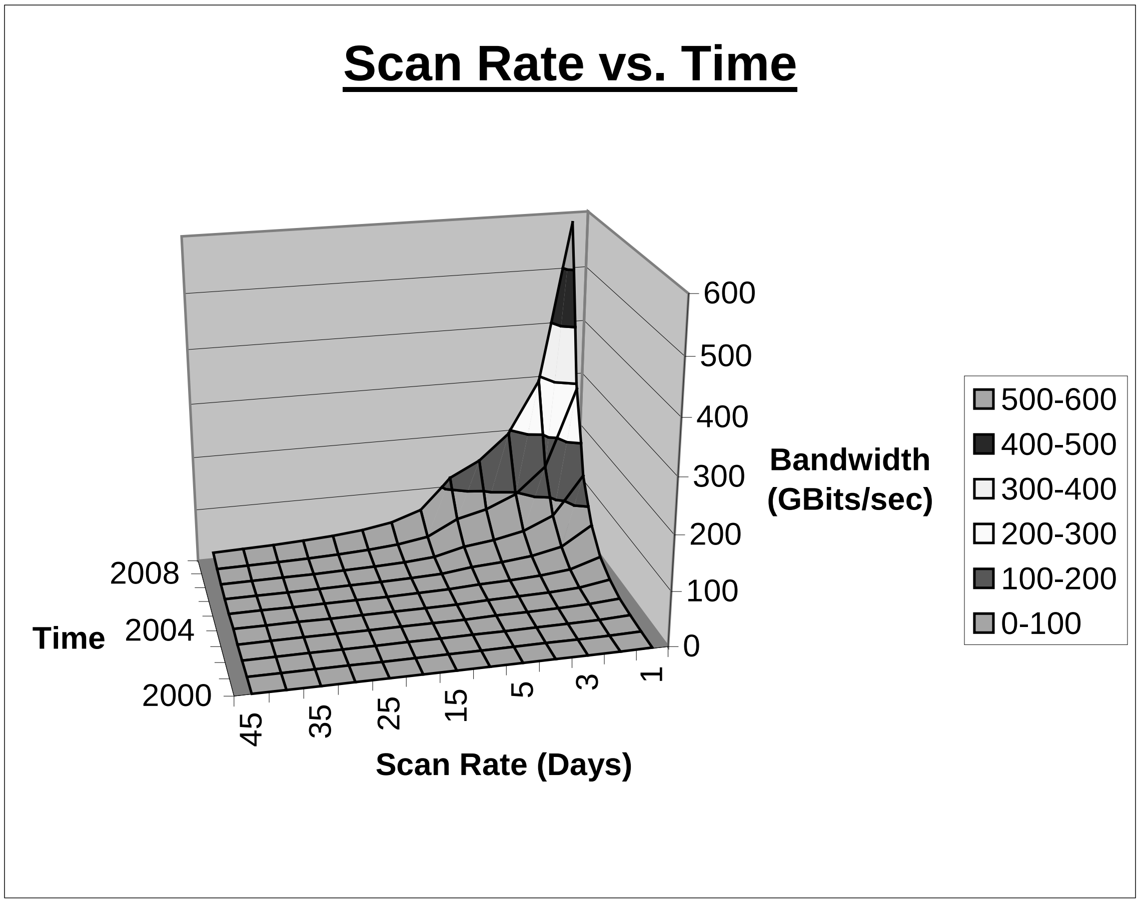


Figure 3: Bandwidth projected by varying time and scan rate

Besides experiencing bandwidth limitations when scanning the WWW, search engines also have the additional problem of managing the cluster of machines necessary to process this large amount of information. If one normal machine can process 1-10 Mbits/sec worth of pages, then achieving a scan rate of 1137Mbits would require a cluster of 114-1137 machines. While managing this type of system is possible, it requires a lot of space, power, and human resources. According to Moore's Law, machines will become more powerful. Power is defined as the combination of resources such as CPU, disk, memory, I/O, etc. Yet this power, does not double every year, and cannot keep pace with the current growth of data on the WWW. The number of machines required to handle the expanding load will continue to increase rapidly, exacerbating the problem.

A new approach to solving this problem is needed. By distributing the crawlers to many points in the network through a **metacomputer**, the crawling algorithm gains some advantages. The HyperDog system was created on this idea and is system described in detail in Section 5. The first advantage HyperDog has is a bandwidth savings to the central indexing point. The crawlers filter the pages they crawl and only return back the pages they perceive as new or changed since their lastvisit. Since only a small portion of the WWW changes each day, the changes sent back to the central indexer are greatly reduced. The crawlers can further save bandwidth by compressing their communications with the indexers. When scan rates are high (~1 day), a savings of ~2 orders of magnitude can be expected.

The second advantage to using a metacomputer system is the ease of manageability and use. The individual machines in a metacomputer are self-managed and administered, and are normally replaced or upgraded on a regular schedule. To the centralized indexer, the metacomputer represents a black-box that feeds it the changes that are occurring on the WWW, without having to manage the crawlers or the infrastructure they run on.

The third reason for using a metacomputer is to find out if this type of architecture is practical. Large metacomputers are relatively new things, so finding the key applications that work well on these systems is important.

HyperDog gives the indexers all the updates that occur without giving them the redundant pages that haven't changed. Since the recrawls are done in a distributed fashion, this allows them to be greatly sped up through parelellization. Initially the speed will be slow because everything will be "new" or "changed" to the crawlers. In this phase the crawlers will be **discovering** the existing content on the WWW. But within a short amount of time, HyperDog will reach a relative steady state, where only the normal growth of the WWW will be reported.

In order to get HyperDog to work, an Internet-wide metacomputer system had to be designed and implemented. This system is the Share Metacomputer. The design of this system flushed out the definitions, types, and fundamental properties of a metacomputer, which are described in Sections 3 and 4.

This report is organized into 4 main sections. Section 2 describes the related work in web crawling and metacomputers. Section 3 examines the fundamental properties of what a metacomputer is and the kinds of jobs that work well on a metacomputer. Section 4 describes the Share metacomputer system, and Section 5 examines the HyperDog system in detail.

2 Background

2.1 Web Crawling Related Work

Web crawling has been done for many years since the early WWW search engines. Most of the major search engines have crawling systems they use to feed their indexers. These include Google [SBLP98, Google], AltaVista [AltV], Inktomi [InkT], and others. Google's system uses an interesting approach of a "random walk" instead of a strict depth-first or breadth-first discovery algorithm. This random walk crawls a URL in a normal depth-first approach. Yet it only continues along that path with some defined probability. Eventually it will abandon that path and start randomly at a new URL. This helps Google avoid infinite looping "crawler traps".

The Informant at Dartmouth [InfoM] is a project that monitors a set of URLs specified by a user and reports via e-mail when one of them changes. It also acts as a meta-search engine and periodically returns the best URLs that match a user's keywords. For both of these options the user has the ability to specify the interval that these checks are performed. From this system, Brewington and Cybenko [BBGC00] were able to confirm their theoretical formula on how dynamic the WWW is. Their formula predicts the bandwidth needs of web crawler based on a desired "up-to-date" rating.

The Mercator system [AHMN99] was developed at Compaq Systems Research Center [CSRC]. It is a web crawling system that was written in Java [Java]. It was written to be very scalable and extensible in what it could do. Their goal was to publicly try to develop and publish a modern web crawling system. Mercator made 77.4 million HTTP requests in 8 days, achieving an average download rate of 112 documents/sec and 1,682 KB/sec. These results where consistent with the results published by many of the major search engines. Unlike HyperDog, Mercator uses a centralized approach with a single machine running within a single network.

2.2 Metacomputer Related Work

The Java Market [Rbor98] is a system where users upload a job via a web site to be completed. These jobs are usually singular in nature and are written in Java. Contributors then point their web-browsers at the Java Market site and a job is automatically started on their machine. The system uses the contributor's web-browser to run the jobs as secured applets.

The Frugal system [Frugal] uses Jini [Jini] technology along with a "Cost-Benefit Framework" to turn any Jini network into a metacomputer. The cost-benefit framework accounts for many parameters (job requirements and system loads) when it makes its decisions on where to place jobs.

The Seti@Home project [SetiH] was developed to search for Extra Terrestrials (ET). The Seti [Seti] project scans the sky for intelligent radio originating outside of the solar system. All the data that is collected is then sent to the Seti@Home system. This system then hands out batches of sky to process for intelligent signals. The batches are handled out to contributor's machines that have downloaded and installed the

Seti@Home client program. The client program runs as a screen saver on the contributor's machine and contacts the Seti servers for work when the machine has idle time.

The Mosix system [BGW93, Mosix] offers a cluster-based metacomputer. Each node in the cluster runs with a modified operating system. Processes are run on any node, and the system can dynamically move jobs around to other nodes in order to most efficiently use the resources of the system. Processes have no knowledge that they have been moved and their execution is not interrupted. Moving a process is done at the OS level.

The Beowulf system [Beowulf] is a set of tools that allows relatively cheap PCs to be connected together to form a cluster. Together with the MPI [MPI] and PVM [PVM] message passing libraries the cluster can become very effective LAN based metacomputer.

The Condor system [Con, LLM88] is a mature system that was first introduced in the 1980s. It requires its jobs to link with special C libraries that replace the standard C libraries. Jobs can checkpoint themselves and write an image of their state to disk. These checkpointed images can be used by the condor scheduler to move jobs from one machine to another. They can also be used to restart a job if it crashes prematurely.

Recently a number of new metacomputer companies have come into existence. These include PopularPower [Ppower], Entropia [Entropia], Parabon [Parabon], and others. These companies are attempting to build large metacomputing networks to run client's parallelized jobs.

A further complete list of related work can be found in [RBor00].

3 Internet-Scale Metacomputer Fundamentals

3.1 *Types of Metacomputers*

There are three main properties that help to classify a metacomputer model.

3.1.1 LAN vs. WAN

A metacomputer can be made up of nodes that are connected on a local area network (LAN) or across a wide area network (WAN). The advantages of a LAN are usually high bandwidth and low latency communications between nodes. A good example of this type of system would be a Beowulf [Beuwulf] cluster or server farm. The advantage to a WAN approach is that the nodes are spread out over many parts of the network, giving better access to the whole network. The nodes in a WAN tend to exist throughout the network graph rather then only at one point. This type of network is also generally more scalable in systems with many nodes (greater then ~1000 nodes). A good example of this type would be the Seti@Home project [SetiH].

3.1.2 Single vs. Multiple Ownership

The nodes of a metacomputer can be owned by one entity or by different entities. When one entity owns all the nodes, the security of the system is greatly enhanced. The disadvantage to this approach however, is that it tends to be very expensive to scale to large numbers of nodes[3]. The single owner has to purchase, upgrade, and maintain all of the nodes. In a multi-ownership environment, the costs of these tasks are distributed over many owners. A single ownership system is generally used when the nodes are dedicated to the task of being in the metacomputer. Multi-ownership systems however, usually arise when the nodes are dedicated to other tasks and only contribute their "spare time" to the metacomputer.

3.1.3 Heterogeneous vs. Homogeneous Nodes

There are two ways of measuring the similarity of machines; their architecture and platform, or the size of the resources they possess. Architecture identifies whether they are running the same type of CPU, motherboard, etc, and platform consists of the operating system, such as Windows, Linux, Solaris, etc. Differences in categories of architecture include x86, Mac, Sun, etc. The size of machine resources can also be used to measure similarity. These resources include CPU clock speed, size of RAM, hard disk, network bandwidth, etc. When nodes have the same architecture and size of resources they are considered homogeneous. Conversely if either architecture or size of resources is not the same they are considered heterogeneous.

[3] Greater then ~1000 nodes

With homogenous nodes the metacomputer has the ability to send jobs to any node in the system. In a heterogeneous system however, jobs cannot necessarily run on all the nodes, or run with optimal efficiency. Special assignment and load-balancing algorithms must be used to effectively use the resources of all the nodes. Heavy workload jobs may need to be processed on strong machines, while low workload jobs can be processed on strong or weak nodes. The homogenous environment tends to make resource allocation easier and more optimal. However creating and managing this system becomes much less flexible, which tends to increase costs. It is much easier to use all kinds of available nodes and build a heterogeneous system, rather then stick with only one machine configuration. Even small changes to a specific model from a vendor can reduce the performance of homogeneous nodes from their optimal.

3.2 Metacomputer Design Considerations

There are a number of challenges that need to be addressed in the design of a metacomputer operating system. Some of the unique ones are outlined below.

3.2.1 Contributor Security

The first problem that comes to people's mind when you ask them to contribute their machine to a metacomputer is "What about security?". Protecting a contributor's node machine is at the heart of any multi-owner metacomputer. The contributing machine can not be damaged, data stored on the machine can't be read, deleted, or changed, nodes behind a firewall can't allow a program running on the metacomputer to violate the security protections provided by the firewall, as well as a number of other things. The metacomputing jobs that run on a node need to be secured from harming the node. One way to fix many of these problems is to use a sandbox approach around the metacomputer job. The Java Virtual Machine provides such a sandbox.

3.2.2 Metacomputer Security

Security also needs to be in place for the controlling servers of the metacomputer. There are a number of things that need to be protected against.

3.2.2.1 Adversarial Breach of Control Communications

This is the classic security problem when it comes to servers. A server needs to secure its communications with its nodes. This prevents an adversary from reading the data, as well as introducing false or corrupted data into the communication stream.

3.2.2.2 Malicious Masquerading

An adversary can masquerade as the controlling servers, and thus gain control of all the metacomputer nodes simultaneously. For example a simple hacking of the server's DNS entry to redirect it to an adversary's server can cause vast damage. Should the entire metacomputer be breached in such a manner, the metacomputer itself could become a very powerful tool in other forms of Internet attack. One such malicious use of the metacomputer could create a "denial of service" attack of the scope never before encountered by orders of magnitude.

When designing the application (Contributor Environment or CE) that resides on a node, the designers need to build in the idea that the controlling servers cannot be fully trusted. Classic authentication should provide reasonable security for this.

3.2.3 Control of a Job on a Contributor Node

The CE that resides on a node needs to have control over the jobs that it runs. It needs to have the ability to start a job, change its priority, kill it, pause it, and monitor and control its use of resources. For example the CE should be able to control how much network bandwidth the job uses, how much disk access it uses, which IP addresses or URLs it can make connections to. While many of these properties are not completely necessary for an operating metacomputer, they provide invaluable fine grain control that helps to keep the contributor's machine safe.

3.2.4 Control of a Job on a Heterogeneous Metacomputer

An interesting challenge that a designer must face is how to manage the global resources of the metacomputer. The system needs to have the ability to run multiple applications at once, each with it's own code that goes out to the nodes. It also needs to try and optimize the use and distribution of these jobs. Network heavy jobs should be placed on nodes with good network access, CPU heavy jobs should be placed on the more powerful nodes, etc. To do this it must know what each of it's nodes capabilities are and what the nodes are currently working on. Some applications may need to take priority over others. If there are 3 jobs running, each requiring 100 nodes, but there are only 200 nodes available on the system, then it must decide how to distribute the jobs to the available nodes.

Things get further complicated when the servers of the client applications require flow control. If application X is running on 100 nodes, and the third party servers that accept results from X, can't handle instructing 100 apps, then the client servers need to instruct some of the apps to stop processing, and tell the metacomputer controllers to stop or slowdown handing out apps of job X.

Optimal load-balancing when multiple resource measurements are used for decision-making, is an important research area. Some very interesting and promising work exists on this topic by Ryan Borgstrom [RBor00].

3.2.5 Public vs. Private Network

When contributors contribute their machines to a metacomputer this opens up a security hole for the contributors in terms of network access. Many contributor machines reside behind firewalls that protect them and other machines on their LAN. Some metacomputer applications would like to access many resources on the public Internet. The problem is restricting the application to only access the publicly available WWW sites, and prevent the application from accessing privileged internal sites that reside behind the firewall on the same LAN.

This is a very difficult problem to solve. There are only two known solutions to date. The first is to enforce a check that any site accessed by a node, must first be cleared by a trusted source outside the firewall. This outside source will validate whether the site is publicly accessible or not. The second is for each LAN to set up a server with knowledge of what is public and private for that LAN. This server is then consulted before a metacomputer application makes a network connection.

3.3 Metacomputer Application Properties

Not all applications work well using an Internet-wide, multiple-owner, heterogeneous node metacomputer paradigm. There are a number of requirements of an application to be a suitable candidate for use in this type of environment.

3.3.1 Embarrassingly Parallel

The application needs to be able to be logically broken up into a large number of pieces. Each of these pieces must be able to be processed in parallel.

3.3.2 Independence of Pieces

Each of the pieces of the applications needs to be essentially independent. One piece can’t depend on the output of another piece. There are many applications that work well on massively parallel supercomputers where one piece depends on the output from another piece. This type of computation will not work well on a wide area metacomputer due to the fact that usually the network latency between nodes is high.

3.3.3 Centralized Control

When a piece of a parallelized application is activated, it should get its directions from a small number of logical sources. This is not a strict rule, but it greatly simplifies the design of the application.

3.3.4 High Processing/Control Overhead Ratio

In order for a parallelized application to cost effectively use a metacomputer, the cost of processing a piece of the application on a node must be less then the cost of processing that piece locally. In order to make this work, the application needs to spend the majority of its time processing the piece of the application on a node, instead of getting that piece of the application to the node, feeding it instructions, and getting results back from it. All the work done not in processing is considered the Control Overhead. This makes up the Processing/Control Overhead ratio. The higher the amount of processing time is, compared to the time needed for control overhead, the more efficient the use of a metacomputer is. The ultimate ratio unfortunately is a single machine with no parallelism. So any efficiency gained by the ratio must be balanced by the parallelism.

3.3.5 Moderate to Low Security Needs

In the multi-ownership metacomputer environment the nodes can be located geographically all over the world, and owned by many different people and entities. This makes the metacomputer not entirely secure. There are a number of ways that it is insecure.

1. If the code or instruction data that is processed by the metacomputer is of vital security, then a multi-owner metacomputer is probably not a good choice. Any node of a metacomputer will eventually fail under a focused attack. For example, the owner of a node in the metacomputer can take a snap shot of the memory of a running node and reverse engineer the application.
2. If the node decrypts any encrypted information, an adversary can decrypt that information as well. This is because not only is the encrypted information available to an adversary, the key is also available if the key is known at any time by the node.
3. Nodes can be fooled into thinking that they are communicating with their servers, even though they are in reality communicating with an adversary masquerading as their server.
4. Servers have no way to guarantee they are communicating with valid nodes, or that the nodes they are communicating with have not been tampered with or altered.

All these security holes make it impossible to give high and ultra-high security guarantees. Yet many of these problems have to be weighed against risk. Reasonable levels and forms of security can be implemented to guard against the majority of threats, and there are many applications that can deal effectively with even these security issues.

3.4 Metacomputer Application Design Considerations

Designing an application for a large-scale metacomputer has many unique problems that do not occur when designing applications for other environments. Some of these unique challenges are:

3.4.1 Integrity of Data

In a metacomputer the data processed by the nodes and the results are of vital importance. Yet because the metacomputer is so expansive, and is made up of so many potentially foreign un-trusted nodes, the integrity of the data is real concern.

3.4.1.1 Data Loss

Data can be lost. When a controller sends out a batch of data to be processed on a node, the node can fail, choose to stop contributing, become overburdened with other jobs that slow down the metacomputing job, or there can be a network communication problem. This essentially means that the processing nodes of a metacomputer are not reliable. Therefore, redundancy and fault-tolerance must be implemented in the application. The two methods often used to add reliability are timeouts and duplication. For the first method, a controller first sends out a job to a node. If the node doesn't return results within a reasonable amount of time, the job is considered lost and is sent out again. The other method is to send out the job more then once, and use the results of whichever comes back first. This speeds up the process. In case one job never returns results, we don't have to wait for the timeout to send the batch out again.

3.4.1.2 Adversarial Conditions and Data Corruption

Data results can be corrupted in a number of ways. There can be an error in communications, or there may be an adversary corrupting the data. An adversary can control a node and therefore change the data in any way. This means that the results from a node can never be fully trusted to be correct. One solution often used to compensate for this is to send out the job twice and compare the results. If they are the same, then the results are assumed to be trusted, if not, then the process can be repeated until the results agree.

3.4.2 Continuous vs. Batched Processing

Developers of metacomputing applications need to make a decision if the application that will run on the nodes of the metacomputer will run continuously, or if it will run, perform a task and then stop. Either method works fine, but if their application uses a start and stop design, then it makes it easy for the metacomputer to move jobs around and load-balance the whole system without losing data. For example, if a running job needs to be moved to another node, then the job can be moved safely once the job completes. But if the job runs continuously, there may not be a logical time to halt and move it. Therefore the job will need to be killed and data could be lost or wasted.

3.4.3 Coordination

Coordinating a metacomputer application can be the hardest design issue. If the application is running on hundreds of thousands of nodes, the controllers that work with the nodes needs to be able to scale to handle them. This is why reducing the complexity and communication between the nodes and the controllers is a good thing. It allows the controllers to scale to larger numbers of nodes. A metacomputer with 10 million concurrent nodes that spend 10% of their time communicating with their controllers means that the controllers need to be designed to instruct 1 million nodes concurrently. This is why a high processing/control overhead ratio is important.

3.4.4 Continuous vs. Sporadic Network Access

The nodes on the metacomputer may have continuous access to the network, or they may have sporadic access. It is important to remember that nodes with sporadic access can still be working on a job even if they don't have an available network connection. In the design of a metacomputer application, the designer needs to take this into account. Once the results of a job are completed, the application may have to wait until there is a network connection to return the results and get new instructions. This of course can look to the controllers as if the job is lost or not responding.

4 The Share Metacomputer

People talk about "The Net" as an amorphous beast. Containing vast amounts of information and having the ability to accomplish many tasks. It is viewed not just as a network of linked resources, but rather as an entity in itself. And to some degree this perception is correct. But the Internet is not yet at the point of mirroring science fiction films where "The Net" itself performs tasks and everyday functions. It is rather a linked system to navigate between varying resources and services.

The idea of a metacomputer brings us closer to the dream of "The Net". A metacomputer is a virtual computer made up of many computers. The hardware of a metacomputer is in place. Today's Internet can be viewed as a connected system of computers that make up the hardware of a metacomputer. What is missing are the two other layers besides hardware that make up a computer: the operating system and the applications.

The Share system is collectively a metacomputer operating system that is made up of many participating computers on the Internet. It consists of two parts. The Contributor Environments (CE) and the Allocation Servers (AS). The CE is a software application that is installed on a contributor's computer. This can be any modern computer with a network connection. The CE runs as a low priority background process. This property attempts to minimize the impact on performance a normal user of the computer will experience. The goal is to harness only the unused resources of the contributor's computer. The task of the CE is to download, monitor, and run jobs given to it by Share.

Completing the system is the Allocation Server(AS). The allocation server coordinates the efforts of all the nodes in the metacomputer. The allocation server or servers, hand out jobs to the CEs, trying to efficiently use the combined resources of all the nodes. It has the ability to add or remove any job running on a node, and to upgrade the CE application to the newest version.

The HyperDog application is one application that has been built to fill in the application layer of Share. HyperDog is discussed in much greater detail in Section 5.

4.1 Assumptions

4.1.1 Controllers and Contributors

The Share system is made up of two components. The Contributor Environment (CE), which runs on contributors machines, and the Allocation Servers that hand out jobs to the CEs. This architecture creates a basic client server model for managing Share.

4.1.2 Jobs Fit The Client Server Model

The applications that run on Share need to fit into the client server model as well. The allocation servers will start a client's application running on a specified number of nodes. Once the app code is running, it is up the client's servers to manage those apps with instructions and data.

4.1.3 Internet-Wide, Multi-Ownership, Heterogeneous

The Share system was designed to be Internet-wide scaling. It was designed to consist of contributing nodes from many people and organizations around the world. These nodes can be of many different types and strengths.

4.1.4 Continuous Node Network Access

It is assumed that nodes in Share have reliable always-on Internet connections. If they don't, it is up to the applications to deal with the problems of sporadic network access. There are no built-in tools to help applications with this problem. Share will still function however with nodes in the sporadic network access paradigm.

4.2 Architecture

Share consists of two parts; Contributor Environment (CE) and Allocation Server (AS). The allocation server controls the system and distributes the metacomputer applications. The contributor environments are the applications that are installed on node computers and run the metacomputer applications from the AS. Together these two parts create the Share metacomputer, which is intended for use on a very large number of nodes. Figure 4 demonstrates how the components of Share work together. Figure 5 shows how this architecture integrates into the global Internet.

Both of these applications have been written in Java2. The CE uses Java2 and it's security mechanisms to run third party jobs and protect the contributor's machine. Using java for the CE also has the advantage of making the CE cross-platform. It can run on Linux, Windows, etc. The AS uses Java because it is a prototype server and Java simplifies the I/O communication between CE and AS. In a production level system, the AS will probably be written in C/C++ for performance reasons.

4.3 Contributor Environment (CE)

4.3.1 System Controller (SC)

The CE has been designed with a number of features to help make the best use of the resources on the node, while maintaining the security of that node. It is comprised of two parts; the System Controller (SC) and the Application Controllers (AC). The SC is the main process of the CE. It always runs, and it is responsible for communicating with the allocation servers. It communicates with an AS after every *Communication _Interval*. The AS will respond to these communications with instructions to the SC. Those responses can be a combination of:

- Launch New Jobs
- Kill Running Jobs
- Upgrade
- Continue / No Change

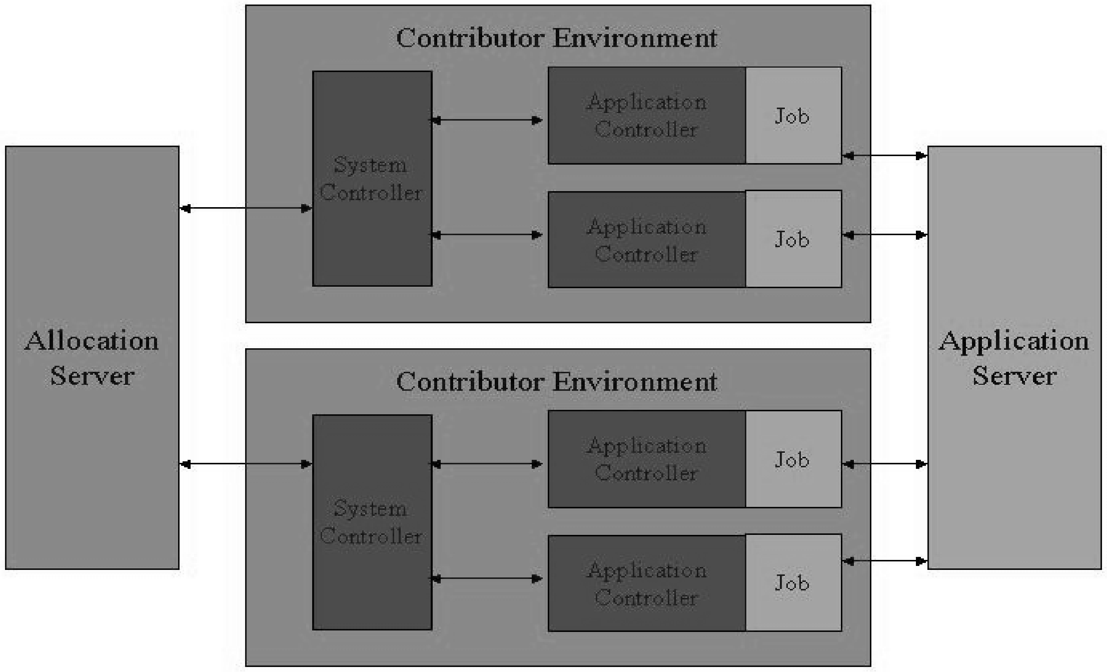


Figure 4: Block Diagram of Share Components

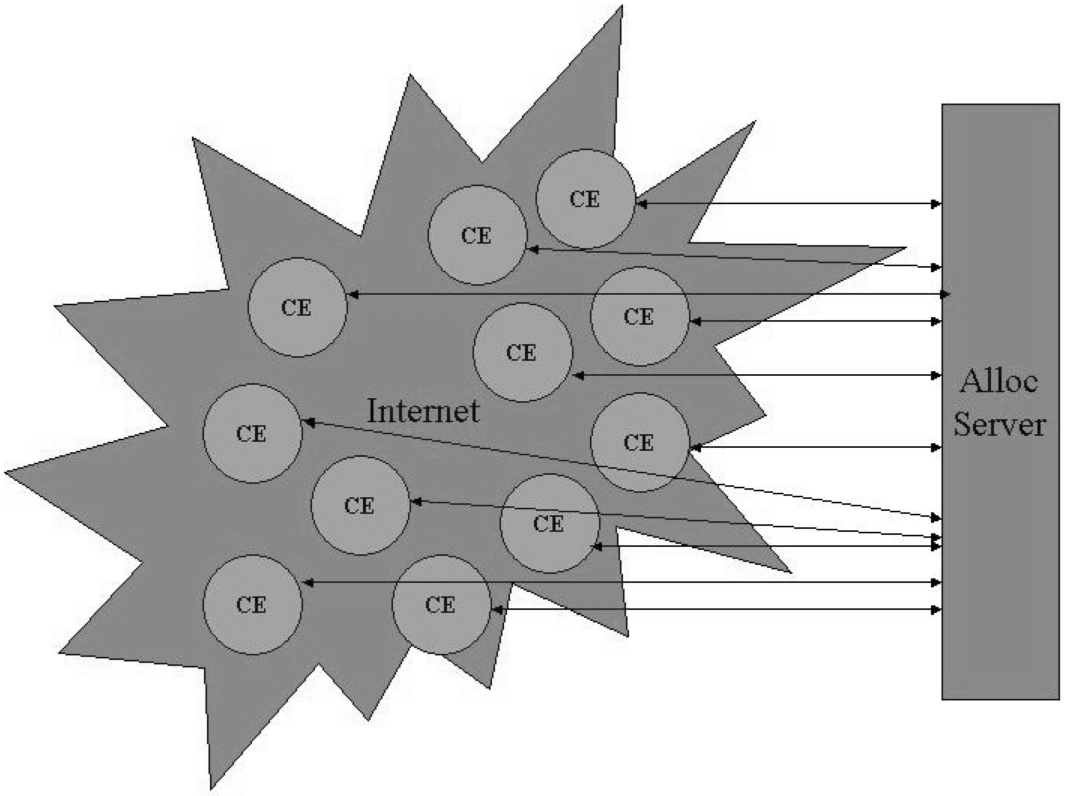


Figure 5: Share system on the Global Internet

To launch a new job, it spawns a new AC process running at low priority, using system specific calls to the operating system of the node. It passes along to the AC process the name of the job to run and where to go get the code for this job. There can only be one SC running on a node, but there can be multiple ACs.

To kill a running job, the SC uses system specific calls to kill the AC process. This helps to guarantee that there are no rouge jobs running on the node that "just refuse to die".

To upgrade itself, the CE first kills all it's running jobs, and then starts the upgrade process. The upgrade process first upgrades the upgrade module. Then the new upgrade module it started in a new process. The main SC then exits. Once this is complete, the upgrade module replaces the old SC with the new SC module. The new SC is then started and the upgrade module exits. This completes the upgrade process.

Communication between the ACs and the SC is accomplished through a local socket connection. When an AC starts up it is given a unique key as one of its parameters. It uses this key to authenticate the SC on the other end of the socket it establishes. The communication socket is then used to perform access checks. Currently no access checks are implemented, but in the future before the job running in the AC can make a function call to a system resource, the request will first be sent to the SC, which checks to make sure the request is reasonable and within the acceptable resource limits. If the check succeeds, the function call completes, if the check fails, an exception is thrown back to the job. This check mechanism tries to ensure that the total resource usage of all the jobs running on a node doesn't overwhelm the node. For example if the contributor gives 10MB of disk space to the CE to run all it's jobs, this check mechanism gives the SC the ability to enforce this restriction across all the jobs, rather then for each individual job. Job A may be given 7MB, and if Job B then requests to use 4MB, it's request will be denied.

4.3.2 Application Controller (AC)

The Application Controller is a wrapper process that runs a metacomputer job. When an AC is created, it is told which job it needs to run and where to get the code for that job. The first thing it does is download this code and then use a special Java ClassLoader to load the class code into a Java Protection Domain. This protection domain restricts the code of the job. This process constitutes the "sandbox model". Once the code is loaded, its primary class is run in a new thread. Once the job is finished running, the AC will exit and notify the SC.

4.4 Allocation Server (AS)

The allocation server is currently in a rough prototype state. Currently it stands alone and cannot work in collaboration with other allocation servers. The AS has a pool of threads that handle connections from CEs. For each of these connections, it passes the information into a decision module that decides what is best for the CE and the global metacomputer. By making the decision module a separate logical unit, it makes the system extensible. Many different decision algorithms can be uses by the system.

4.5 Future Work

There are many areas of improvement for the Share system. Some are engineering problems, efficiency optimizations, and others. Below are also some next steps to the architecture.

4.5.1 Dynamic Clustering

All the nodes are currently considered to be in the amorphous cloud of the Internet. Yet there are groupings of nodes that reside on the same network which give nodes low latency and high bandwidth communications between each other. Finding these clusters of nodes dynamically would be very helpful. It gives a metacomputer the ability to deliver a dynamic set of nodes that can be configured to run as a cluster. These nodes could inter-communicate with each other, whereas currently that is discouraged because of latency, bandwidth, and control problems. In this way, a metacomputer that allocates single machines as a resource, can evolve into one that allocates dynamically generated work-clusters.

4.5.2 Out-Of-The-Box Solution

Share has been designed to work on the Internet-wide scale. Once that is accomplished, the system can be scaled down to work in an enterprise environment. This would provide many advantages. It would help alleviate many security problems and it would provide better bandwidth and latency guarantees, while maintaining the heterogeneous flexibility of the Internet-Wide Share system.

5 HyperDog

HyperDog has been designed to monitor the state of the WWW using a metacomputer such as Share. It visits each URL available as often as resources will allow. It consists of two levels. The first level is the crawlers. These are the applications that run on the many nodes of the metacomputer. To fit into the Share contributor environment they were written in Java2. They track how pages change, find new pages, and find deleted pages. The second level is the backend data-center. The data-center is responsible for coordinating the crawlers and compiling the results they send back. It consists of a distributed system of **linkservers**, the **database**, **database bridge**, and the **linkadmin**. The linkservers manage the communication with the crawlers and the tracking of URLs. The database bridge program knows how to handle updates from the linkservers and also how to work with the database. The database is used for data mining and linkservers use it for data recovery when a linkserver crashes. The linkadmin program is a user-interface to HyperDog that allows an administrator to monitor and control the state of HyperDog. All of the components of the data-center have been written in Java2 to speed up the development process and for easy I/O communication. Figure 6 shows how the components of HyperDog and the data-center interact. Figure 7 shows how HyperDog's components function in the metacomputer environment.

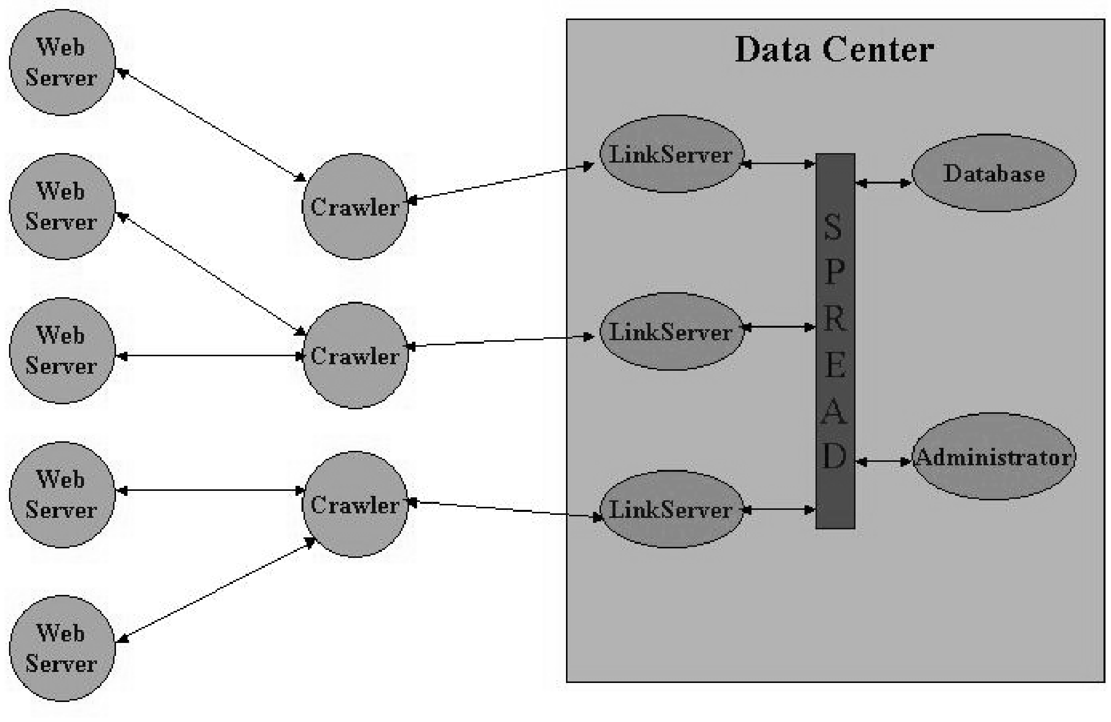


Figure 6: Block Diagram of HyperDog Components and Architecture

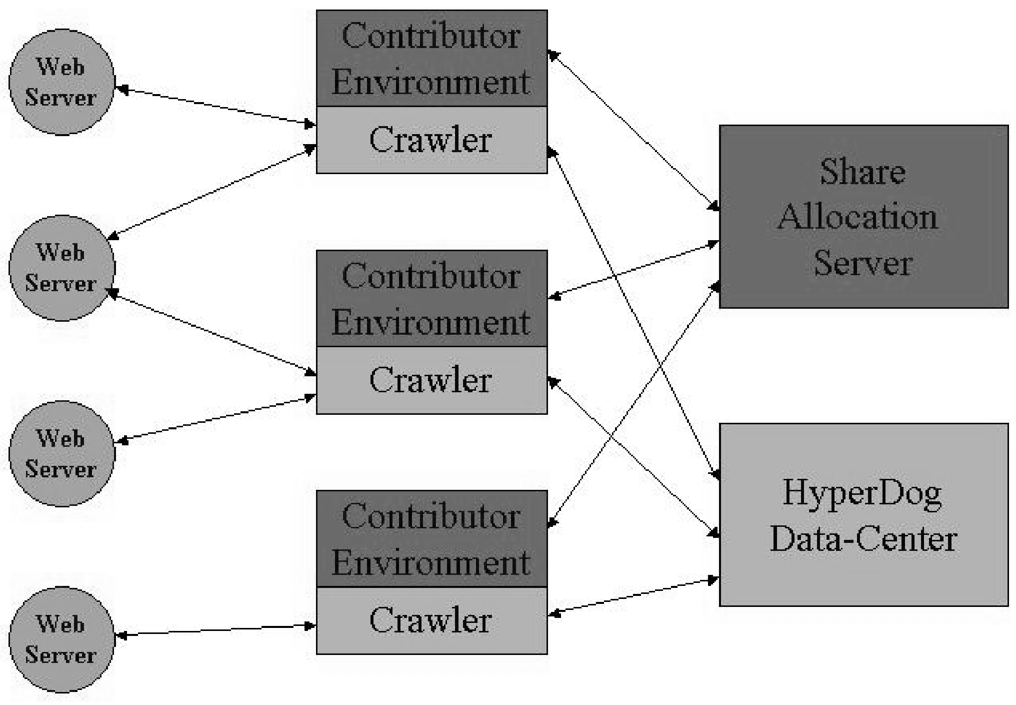


Figure 7: Interaction of HyperDog components in Share

5.1 Crawlers

The crawlers contact the linkservers to obtain a batch of URLs to check. The linkservers will either return a batch of URLs, direct the crawler to try again after some delay to a new location, or will instruct the crawler to stop and exit. When a batch of URLs is successfully sent out to a crawler, it is processed, and the results are returned. This process continually repeats. Since conserving bandwidth to and from the linkservers is one of the primary advantages of this architecture, all communication between the crawlers and linkservers is compressed. To avoid large discrepancies in timestamps, the crawlers maintain a pseudo-synchronized clock relative to the linkservers[4]. All crawlers obey the Robots Exclusion Protocol [RBP] contained in each site's robots.txt file.

A batch contains a number of bundles of URLs. Each bundle contains URLs from the same domain. The crawler spawns off a separate thread to handle each of the bundles contained in the batch. The number of bundles is usually limited to ~30-100 per batch, each containing ~100 URLs. For each URL the linkserver sends the full URL address, the date the URL was last checked, and a 64-bit hash of the URL's page content. Each thread that gets spawned off performs the following algorithm on its URLs. Figure 8 displays the states the algorithm goes through.

[4] The system time of the crawler is not changed. Whenever a crawler creates a time stamp, it corrects its time to reflect the linkserver's time.

1. Verify that the domain’s robot.txt file allows access to the URL’s content.
2. Connect to the URL and get it’s header information.
3. If the URL cannot be contacted, or an I/O exception occurs anywhere in this algorithm, the URL is assumed to be dead. The crawler then reports back to the linkserver that this URL is dead. After a URL is declared dead a set number of times, it is removed from HyperDog.
4. The URL’s page type, size, language, etc, are recorded.
5. If the page has a HTTP *Last-Modified* date field in its header, the crawler checks to see if the page has changed since the last visit.
6. If it has not changed then it continues on to the next URL. If the page has changed, or the *Last-Modified* date is not reported, then the algorithm proceeds on.
7. If the HTTP *Content-Type* is “text” or “html” then it’s content is downloaded. Else the algorithm continues on to the next URL of the bundle.
8. The new hash code for the downloaded page is calculated. If the new hash code and the previously recorded hash codes are the same, then the page hasn’t changed, and the algorithm continues on to the next URL. If the hash codes are different, then the page has changed.
9. The page information to be returned to the linkserver is updated only if the page has been classified as changed. This updated information includes the new changed date, the new hash code, etc, along with a list of all the URLs that were found on changed pages.
10. For the changed pages, the valid html URLs are extracted and returned with the page information to the linkserver. Non-valid URLs include malformed URLs, URLs that do not use the “http” or “https” protocols, etc. URLs that contain parameters are stripped of their parameters to try and prevent dynamic content URLs from entering HyperDog.
11. The algorithm then repeats on the next URL in the bundle. Before the next URL is examined, the crawler first makes sure that a ½ second of time has elapsed since the start of the algorithm on this URL. This ensures that the web-server is not overloaded.
12. All information regarding this batch, which needs to be sent back to the linkserver, is batched up and sent as one large compressed package. This saves bandwidth, and increases the speed at which crawlers can process URLs.

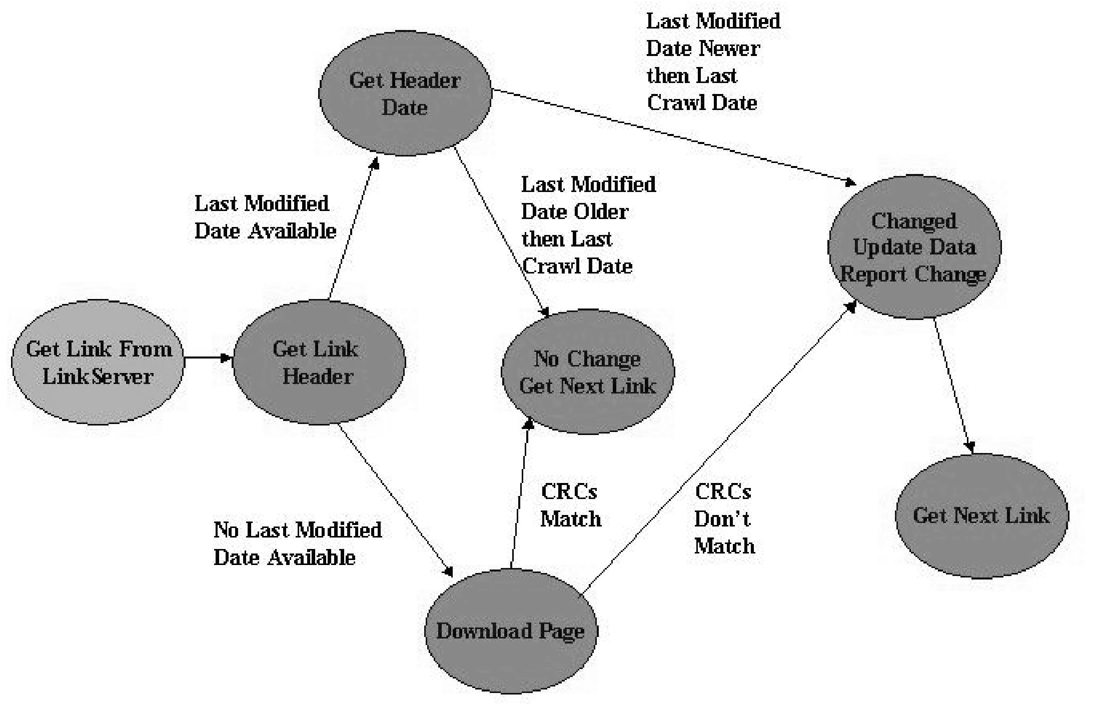

Get Link From LinkServer
Get Link Header
Last Modified Date Available
Get Header Date
Last Modified Date Newer then Last Crawl Date
Last Modified Date Older then Last Crawl Date
Changed Update Data Report Change
No Change Get Next Link
No Last Modified Date Available
CRCs Match
CRCs Don't Match
Get Next Link
Download Page

Figure 8: State Diagram of The Crawler Algorithm

5.1.1 Implementation Issues

A few problems were encountered in the implementation of the crawler. The first was with the parser used to extract the URLs from an HTML page. The crawlers use the javax.swing.HTMLDocument's parser feature to do this task. This object however uses a lot of memory and needs to connect to an X-server when running under Linux. For this reason a new parser will be used in later versions. The other major problem that was found was in Java's HTTPConnection class. There is no time-out method available for this class. When HTTP requests are sent to a webserver, not all webservers answer correctly. If the webserver keeps the underlying TCP/IP connection alive, but refuses to send the response to an HTTP request, the crawler will hang. There does not seem to be a way to prevent this, so a new HTTPConnection class will have to be written in later versions.

5.2 Data-Center

5.2.1 LinkServers

Managing information on 2 billion URLs that get crawled every day is a tough design problem. A single linkserver cannot accomplish this task. Therefore, the problem of managing information on all these pages has been split-up into a distributed system of linkservers. Each linkserver knows about the other linkservers. The data is split-up between these linkservers by URL domain. Each linkserver keeps a record of which linkservers are participating, and it knows which linkserver is responsible for which domains. Within a domain however, the data is only stored on a single machine. By splitting the data up to multiple machines, HyperDog can handle the large number of URLs and can scale effectively with the growth of the WWW. (See Figures 1, 2, & 3)

The linkservers were written in Java2. This makes HyperDog flexible enough to run on clusters made up of different types of machines. It also simplifies the I/O communications between linkservers and crawlers.

The linkservers are composed of several subsystems. The crawling subsystem handles the communication with crawlers. It receives requests for batches of URLs and then queries the queuing subsystem for a batch. The queuing subsystem is responsible for tracking the state of all bundles in the system. The Spread subsystem is the communications layer between the linkservers. It is responsible for keeping the linkservers in sync with each other. The load-balancing subsystem balances the domains of URLs evenly throughout the set of linkservers. It uses the transfer and acceptor subsystems to help accomplish this.

5.2.1.1 Crawling Subsystem

The linkservers accept connections from crawlers. DNS Round Robin is used to distribute the connections evenly over the linkservers. Later, this will be combined with more sophisticated load-balancing technology, such as Backhand [BackH]. Each linkserver maintains a pool of threads to handle the crawler connections. For each new connection, the linkserver queries its queues of bundles of URLs to see which need to be crawled next. It identifies these top N bundles and sends them out to the crawler to be checked. If the linkserver does not have any bundles to give out, it instructs the crawler to return later.

When crawlers return the results from a bundle, the linkservers need to process these results. The first thing a linkserver does is check if it is the owner of the URL domain for that bundle. If it is the owner, it updates its data on the URLs in the bundle. If it is not the owner then it sends that bundle's results on to the linkserver that is the owner. It then processes all the new URLs that are returned with this batch. For those URLs that are in the domains that this linkserver owns, it creates new records for these URLs. For the URLs in domains it doesn't own, it sends them on to the linkservers that do own them. For URLs in domains that are not know by HyperDog yet, the linkserver determines if the domain is allowed, and then it announces to the group of linkservers that it has found a new domain. The linkservers synchronously arrive at the same decision on the owner of the domain. The new owner then creates the new domain and inserts the new URL into its records. When new URL records are created they are added to bundles until the bundle is full. All bundles have a fixed size set at the initialization of the system. Currently the bundle size is set to 100. When dead URLs are removed from bundles, the space is replaced by URLs at then end of the last bundle for this domain. This ensures that all bundles, except at most one, are of equal size and the bundles do not become fragmented.

URLs are removed from HyperDog when they have been checked a certain number[5] of consecutive times and each time the crawler reports that the URL is dead. This helps to ensure that only URLs that are alive remain in HyperDog.

5.2.1.2 Queuing Subsystem

Each linkserver maintains responsibility over many bundles of URLs. These bundles need to be rechecked by crawlers after every period of time defined by the desired scan rate. To accomplish this, linkservers maintain a queuing system of bundles. There are three queues or states that a bundle can be in.

- **Normal**: In this state, a bundle is waiting to go out to a crawler. It has a time stamp of when it needs to go out by, but it has not yet reached this scheduled time.
- **High**: In this state, a bundle had been sent out and placed in the pending state, but there were no results for it. Therefore it is overdue to go out, and placed in this state.
- **Pending**: In this state, the bundle has been given out for a crawler to check. The linkserver is waiting for the crawler to return with results.

When a linkserver needs to give out bundles, it gives out the top N bundles. These bundles are composed of the top N bundles from the *high* priority state, and the remaining bundles if needed from the *normal* priority state. Before a batch of bundles is compiled, all the bundles that have been in the *pending* state for too long are moved into the *high* priority state. By moving bundles that have been in the pending state for too long into the high priority state, the linkserver ensures that all bundles get checked.

As bundles return, they are removed from the pending state and placed in the normal state. The time that they are scheduled for rechecking is randomly chosen between now and a max amount of time T equal to the desired scan rate of HyperDog. The reason that the time is randomly chosen is to give an even distribution of when bundles go out. When bundles only go out at their scheduled time, randomization is required, or there will be large clumpings of bundles that need to go out at one time, and nothing in between. This method tries to ensure continuous smooth use of the resources available to HyperDog.

This queuing subsystem makes a best effort to make sure all the bundles are rechecked during every scan. There are two methods that are used to do this. Either it will always give out bundles in the high or normal states if it has any (regardless of when they are scheduled to go out), or it will only take bundles that have reached their scheduled time to be crawled. The normal method of operation

[5] Currently set at 5

is to take the top bundles, regardless of scheduled time. In this case the scheduled times are used solely for ordering purposes. This method keeps the system scanning the links as fast as it has resources to do so. The other method will maintain the desired scan rate using the scheduled times if HyperDog has enough resources and is not behind schedule. This method will be used in later versions of HyperDog when the scan rate is different depending on the bundle or domain.

5.2.1.3 Spread Subsystem

Communication between the linkservers and the other backend components is crucial. All the linkservers need to keep a common state of which linkservers are responsible for which domains. They also need to decide what to do when changes occur, such as machines come up, go down, new domains are found, domains need to be moved, etc. In order to keep the linkservers in sync with each other, the Spread Group Communication Toolkit [AS98, Spread] is used. The Spread system provides a reliable group multicast overlay-network for the backend servers to communicate on. HyperDog uses three Spread groups:

1. **LinkServer Group**

   The linkservers use the linkserver-group and the *Agreed* property to send messages to the group of linkservers. This property ensures that all linkservers that are alive will receive all the messages in the same order. This is the key property that allows the linkservers to be synchronized and to act together as if they were logically one system. They use the membership services of Spread to identify the members of the group, and to receive notification of member joins, leaves, and crashes.

2. **Admin Group**

   The linkadmin uses the admin-group to send command to the linkservers. The responses are sent back to the linkadmin on it's private group.

3. **Stream Group**

   Linkservers broadcast all the changes they find on the WWW to the stream-group. Databases and other data-mining servers listen to this group for these updates.

5.2.1.4 Load-Balancing

HyperDog load-balances URLs in the system across all of the linkservers. This ensures that no linkserver is more overloaded then any other linkserver. To do this, HyperDog needs to be able to migrate data from one linkserver to another without losing data or interrupting performance. For this to occur, the transfer subsystem and acceptor subsystems discussed later where created.

The load-balancing algorithm[6] gets triggered when the number of URLs in a domain grows or shrinks by a power of 2. All domains start out at a size of 128 URLs. When the size grows to 128, the load-balancing algorithm is called to find the optimal place in HyperDog for this domain. The new trigger points for load-balancing this domain are then set at 2*128 and 128/2. If the 256 trigger point is reached, the load-balancing algorithm is called again for this domain, and it's new trigger points become 2*256 and 256/2. This process continues while the domain is in HyperDog, but the trigger point is always lower-bounded by 128.

Load-balancing is needed because domains come in very different sizes. Some domains have very large numbers of URLs like www.yahoo.com, while other domains only have a few URLs like www.cnds.jhu.edu. A linkserver can either handle a few large domains, or many small domains. The load-balancing algorithm tries to take this into account and provide a roughly even distribution of URLs for all of the linkservers, scaled to what that linkserver has resources enough to handle. Currently the only metric that is being used to determine the best location for a domain when it reaches a trigger point is memory size. Other metrics will be added in later to take into account multiple data-centers of linkservers that are separated by geography or high network latency. In such situations, preference will be given to servers located at the same site.

5.2.1.5 Transfer and Acceptor Subsystem

The transfer subsystem is responsible for moving a domain from one linkserver to another when the load-balancing algorithm decides a transfer is needed. When the load-balancing algorithm is called, each member of the group calculates to which linkserver the domain should go to. Since all of the linkservers are working from a synchronized state, they will all come to the same conclusion. If a transfer is needed, each linkserver knows whom the old owner of the domain is, who the new owner is, and that the domain is now *in transit*. The new owner then instructs the acceptor subsystem to listen for the new domain. The old owner continually tries to send the domain to the new owner until the transfer is complete or the new owner leaves the group. If the new owner leaves the group, the group collectively decides who will receive the domain now, and the old owner starts the transfer to the "new" new owner (if such a transfer is needed).

Once the new owner receives a complete copy of the domain, it informs the group that it is now the true owner of the domain. Consequentially, all the linkservers remove the domain from its *in transition* state. Results for the domain that arrived while the domain was *in transit* were queued, and are now applied.

[6] Suggested by Ryan Borgstrom, http://www.cnds.jhu.edu/~rsean

5.2.1.6 Adding LinkServers and LinkServer Reliability

The backend system of linkservers was designed to be very flexible. New linkservers can be added at anytime, and HyperDog will continue to run when servers are taken offline or when they crash. When new linkservers come up, they join the linkserver's Spread communication group. If the new linkserver sees that it is the only linkserver in the group, then it boot straps itself and starts running. If there are other linkservers in the group then the new linkserver announces to the group that it wants to participate. It then queues all messages until it has been synchronized with the other linkservers. It opens up an acceptor thread that waits to receive the synchronized state information. The oldest member of the group sends the synchronized state information to the new server. Once the oldest member sends this information, it announces to the group that the new server is now up and running. The new server takes this new information as it's state, and starts running. It first applies all the messages it was buffering to this state and then continues running normally.

When a linkserver leaves or crashes, the Spread toolkit notifies the other linkservers of the group that the member has left. They know which domains that linkserver was responsible for and each server then calls the load-balancing algorithm on these domains. They then collectively decide what to do with each of the lost domains. Since they are running from the same view of the system, and use the same load-balancing algorithm, they will all come to the same conclusion for the domains. Each server that gets new domains from this algorithm is now responsible for those domains. A linkserver responsible for a new domain will attempt to recover the URLs that were lost by querying the central database that may or may not have a record of them. If it does, then the database does a transfer to the linkserver, else the linkserver just seeds the domain with a root URL and continues on. The URLs lost will quickly be recovered through the normal discovery/crawling process.

5.2.1.7 The Stream

Changes that are found on URLs are automatically broadcasted to the stream group. This is called the **stream of updates**, or, simply the stream. These messages are useful for non-linkserver members of the group such as indexers or database bridges. The updates that they receive will be new URLs, notification that a URL has changed, or notification that a previously alive URL is now considered by HyperDog to be dead.

5.2.1.8 Limiting the Domains

It is necessary to limit the domains in HyperDog in order to effectively use the resources allocated and to target specific domains or exclude domains. For example, the administrator of HyperDog my wish to track all .edu domains, but exclude all other domains. HyperDog can allow or disallow a specific domain, or it can allow or disallow any domains that match a pattern, for example, any domain that "ends with .edu".

5.2.2 Database and Database Bridge

The database bridge listens to the stream of updates and records this information in the database. The database bridge can then be used by the linkservers to recover data when a server crashes. The database can also be used for data-mining purposes and for gathering statistics on HyperDog as well as the network, domains, and URLs HyperDog is tracking. Specifically, linking information is stored in the database. With this information a complete graph can be assembled of how the static WWW pages link together. By listening to the stream for a while an application can generate a graph of how the WWW is linked and how this linking changes with time.

The problem with this database system is that most databases cannot keep up with the enormous volume of changes HyperDog generates at full speed. In house experiments of Microsoft SQL Server 7.0 [MSSQL] and MySQL [MySQL] have shown that the highest number of inserts into a simple table is ~800/sec for SQL Server and ~2000/sec for MySQL. None of these solutions (running on a normal PC) can keep up with thousands of crawlers and tens of linkservers. The solution to this problem has been left for later. To ensure data recovery, the next step in implementation will be to write these updates to disk. This relatively simple solution will be able to keep up with the updates, until an appropriate database solution can be found. This provides the basic data recovery property needed by the linkservers.

5.2.3 LinkAdmin

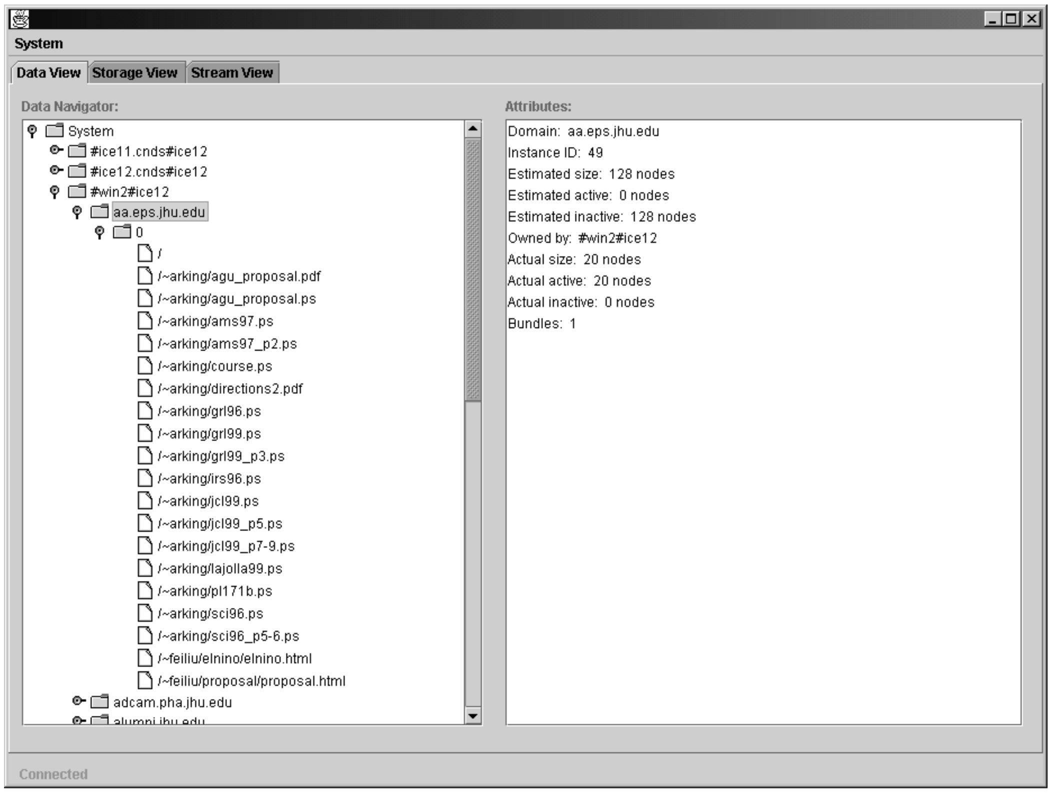
A graphical administrative interface was designed for HyperDog for two reasons. The first was to troubleshoot and debug HyperDog during testing. The other was to present an easy-to-use interface to control HyperDog. The interface program joins the Spread group and is able to send commands to HyperDog. The relevant linkservers will respond to requests from the linkadmin program.

There are two main views of the UI. The **Data View** allows the user to see all the linkservers currently running in a tree hierarchy. Expanding a linkserver shows all the domains a linkserver is responsible for. Expanding a domain, displays all the bundles for the domain. Expanding each bundle produces the URLs that are contained in a bundle. By selecting on these objects, the user gets information on the object. The URL displays the URL name, size, file type, etc. Information on a bundle shows the state of the bundle (High, Normal, Pending), when it is scheduled to go out, etc. Domains display their size, location, trigger points, etc. The linkserver information shows, how many domains a linkserver has, how much memory it is using, it’s load, etc. General system information is available such as bundle size, scan rate, etc. Actions can be performed on these objects.

- Domains can be added to HyperDog or to a specific server
- HyperDog can be shutdown, or a specific linkserver can be shutdown
- Domains can be moved from one linkserver to another
- Domains and URLs can be removed from HyperDog
- The user can set the recrawl time for a bundle

- Domains can be allowed or disallowed from HyperDog
- Entire domains can be cleared of their data and reseeded with the root URL

The second view is the **Stream View**. This view displays in real-time all the URL addresses that are found to have changed. They scroll by in real time as the change notifications are broadcast to the stream. This UI gives the user fine-grain control over HyperDog and allows them to manage all the linkservers and HyperDog from a single point.



```
System
  #ice11.cnds#ice12
  #ice12.cnds#ice12
  #win2#ice12
    aa.eps.jhu.edu
      0
        /
        /~arking/agu_proposal.pdf
        /~arking/agu_proposal.ps
        /~arking/ams97.ps
        /~arking/ams97_p2.ps
        /~arking/course.ps
        /~arking/directions2.pdf
        /~arking/grl96.ps
        /~arking/grl99.ps
        /~arking/grl99_p3.ps
        /~arking/irs96.ps
        /~arking/jcl99.ps
        /~arking/jcl99_p5.ps
        /~arking/jcl99_p7-9.ps
        /~arking/lajolla99.ps
        /~arking/pl171b.ps
        /~arking/sci96.ps
        /~arking/sci96_p5-6.ps
        /~feiliu/elnino/elnino.html
        /~feiliu/proposal/proposal.html
    adcam.pha.jhu.edu
    alumni.jhu.edu
```

```
Domain:  aa.eps.jhu.edu
Instance ID:  49
Estimated size:  128 nodes
Estimated active:  0 nodes
Estimated inactive:  128 nodes
Owned by:  #win2#ice12
Actual size:  20 nodes
Actual active:  20 nodes
Actual inactive:  0 nodes
Bundles:  1
```

Figure 9: Screen Shot of the LinkAdmin Program

5.3 *Future Work*

Two immediate extensions to HyperDog are planned. The first is to add the ability for multiple backend data-centers. Currently all linkservers communicate over a high-speed low latency LAN. By placing data-centers closer to the URLs that they track, the system can gain increased performance. Metacomputer nodes running the crawler would contact the closest data-center to them. These nodes would then crawl URLs closer to them, rather then crawling URLs that are latency-wise far away. A cluster of linkservers would be located in China to track the majority of Asian websites. European and US based data-centers would work the same way. Each data-center would still have to keep it’s state synchronized however between linkservers regardless of location. Domain moves however would be skewed to discourage moves between data-centers.

The second change would be to gather history about the change rate of a bundle. Bundles with URLs that change more often would be scheduled for recrawl more often then bundles that primarily contained URLs that never change. This would constitute a better use of the available resources in HyperDog.

6 Conclusions

The growth of the WWW is explosive and HyperDog represents a cost effective way to handle this growth. It's flexible and immensely scalable design gives it the ability to match this explosive information growth curve. It's scalability is derived from the use of many crawlers on large global metacomputers and through a scalable distributed system of linkservers. We described a need for HyperDog and how it works.

In order for HyperDog to be built, a metacomputer was needed. We examined the types of metacomputers and the design considerations for building one. Through this discussion, the properties of applications that work in such an environment were also examined. Internet-wide, multi-owner, heterogeneous metacomputers provide an extremely powerful resource for those applications that can harness the environment correctly.

7 References

[AHMN99] Allan Heydon and Marc Najork, **Mercator: A Scalable, Extensible Web Crawler**, June 26, 1999, http://www.research.digital.com/SRC/mercator/papers/www/paper.html

[AS98] Yair Amir and Jonathan Stanton, **The Spread Wide Area Group Communication System**, 1998, http://www.cnds.jhu.edu/publications/

[AltV] http://www.altavista.com/

[BackH] Theo Schlossnagle, **The Backhand Load-Balancing Project**, http://www.backhand.org/

[BBGC00] Brian E. Brewington and George Cybenko, **How Dynamic Is The Web?**, Dartmouth College, In Proceedings of the Ninth International World Wide Web Conference, May, 2000.

[Beowulf] http://www.beowulf.org/

[BGW93] Barak A., Guday S. and Wheeler R., **The MOSIX Distributed Operating System, Load Balancing for UNIX**, Springer-Verlag, 1993

[Con] http://www.cs.wisc.edu/condor

[CSRC] http://www.research.digital.com/SRC/

[Entropia] http://www.entropia.com

[Frugal] http://www.cnds.jhu.edu/projects/frugal/

[Google] http://www.google.com

[InfoM] http://informant.dartmouth.edu/

[InkT] http://www.inktomi.com/

[Java] http://www.sun.com/

[Jini] http://www.jini.org/

[LLM88] M. Litzkoow, M.Livny, M.W. Mutka, **Condor – A Hunter of Idle Workstations**, the 8th International Conference of Distributed Computing Systems, June 1988.

[Mosix] http://www.mosix.org/

[MPI] http://www.mpi-forum.org/

[MSSQL] http://www.microsoft.com/sql/default.htm

[MySQL] http://www.mysql.com/

[Parabon] http://www.parabon.com

[Ppower] http://www.popularpower.com

[PMV] http://www.epm.ornl.gov/pvm/pvm_home.html

[RBor00] Ryan Borgstrom, **A Cost-Benefit Approach to Resource Allocation in Scalable Metacomputers**, Ph.D. thesis, Department of Computer Science, Johns Hopkins University, 2000, http://www.cnds.jhu.edu/publications/

[Rbor98] Ryan Borgstrom, The Java Market: Transforming the Internet into a Metacomputer, 1998, http://www.cnds.jhu.edu/projects/metacomputing/

[RBP] Robots Exclusion Protocol, http://info.webcrawler.com/mak/projects/robots/exclusion.html

[SBLP98] Sergey Brin and Lawrence Page, **The Anatomy of a Large-Scale Hypertextual Web Search Engine**, the 7th International World Wide Web Conference (WWW7), 1998.

[Spread] http://www.spread.org/

[Seti] http://www.seti.org/

[SetiH] http://setiathome.ssl.berkeley.edu/

[Sew] Danny Sullivan, **Search Engine Sizes**, http://www.searchenginewatch.com, July 7, 2000.