7-2002

Content Aware Request Distribution for High Performance Web Service: A Performance Study

Robert M. Jones
Portland State University

THESIS APPROVAL

The abstract and thesis of Robert M. Jones for the Master of Science in Computer Science were presented July 12, 2002, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

_____
Karen Karavanic, Chair

_____
Warren Harrison

_____
Robert Bertini
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

_____
Cynthia Brown, Chair
Department of Computer Science

ABSTRACT

An abstract of the thesis by Robert M. Jones for the Master of Science in Computer Science presented July 12, 2002.

Title: Content Aware Request Distribution for High Performance Web Service: A Performance Study

The World Wide Web is becoming a basic infrastructure for a variety of services, and the increases in audience size and client network bandwidth create service demands that are outpacing server capacity. Web clusters are one solution to this need for high-performance, highly available web server systems. We are interested in load distribution techniques, specifically Layer-7 algorithms that are content-aware. Layer-7 algorithms allow distribution control based on the specific content requested, which is advantageous for a system that offers highly heterogenous services. We examine the performance of the Client Aware Policy (CAP) on a Linux/Apache web cluster consisting of a single web switch that directs requests to a pool of dual-processor SMP nodes. We show that the performance advantage of CAP over simple algorithms such as random and round-robin is as high as 29% on our testbed consisting of a mixture of static and dynamic content. Under heavily loaded conditions however, the performance decreases to the level of random distribution. In studying SMP vs. uniprocessor performance using the same number of processors with CAP distribution, we find that SMP dual-processor nodes under moderate workload levels provide

equivalent throughput as the same number of CPU's in a uniprocessor cluster. As workload increases to a heavily loaded state however, the SMP cluster shows reduced throughput compared to a cluster using uniprocessor nodes. We show that the web cluster's maximum throughput increases linearly with the addition of more nodes to the server pool. We conclude that CAP is advantageous over random or round-robin distribution under certain conditions for highly dynamic workloads, and suggest some future enhancements that may improve its performance.

CONTENT AWARE REQUEST DISTRIBUTION FOR HIGH PERFORMANCE WEB SERVICE: A PERFORMANCE STUDY

by

ROBERT M. JONES

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
2002

Acknowledgments

I would like to thank fellow student Richa Sehgal for her assistance with the Webstone benchmarking package and research on workload characterization. Many thanks go to my advisor, Dr. Karen Karavanic, for her patient guidance, candid assessments and thought-provoking questions. Her flexibility and understanding allowed me to juggle both school work and my full-time employment. I would also like to acknowledge Dr. Warren Harrison and Dr. Robert Bertini who participated on my thesis defense committee. Their time and effort in reviewing my work and providing valuable input are most appreciated. Of course, I must thank my family most of all, for without their never ending support, my return to school and completion of the masters program and this thesis research would not have been possible.

Table of Contents

List of Figures iv

List of Tables v

1 Introduction 1

1.1 Popular Website Workloads 1

1.2 Motivation for High-Performance Web Service 2

1.3 Web Workload Characterization 2

1.4 Web Server Architectures 3

1.5 Load Distribution and Balancing in Web Clusters 5

1.6 Load Distribution Algorithms 7

1.7 Contributions of this Research 8

1.8 Summary of Results 9

1.9 Outline of this Paper 10

2 Related Work 11

2.1 Locality Aware Request Distribution Policy (LARD) 11

2.2 Client Aware Policy (CAP) 12

2.3 Web Content Workloads 13

2.4 Web Switches 14

2.4.1 TCP Splicing 14

2.4.2 TCP Handoff 15

2.4.3 Reverse-Proxy Forwarding 15

Table of Contents (continued)

2.5 CAP and LARD Performance 16

3 Experimental Design 19

3.1 Testbed Architecture 19

3.2 Testbed Software Details 20

4 Experimental results 22

4.1 Capacity of the Reverse-Proxy Web Switch 22

4.2 Static Workloads 23

4.3 Dynamic Workloads 26

4.3.1 80% Static, 20% Dynamic Workload 28

4.3.2 60% Static, 40% Dynamic Workload 29

4.4 SMP versus Uniprocessor Nodes 34

4.5 Scalability 36

5 Conclusions and Future Work 38

6 References 41

List of Figures

Figure 1: Example Web Cluster. 4

Figure 2: Capacity of the reverse-proxy web switch. 22

Figure 3: Static Workload Results 25

Figure 4: 80% Static, 20% Dynamic Workload 28

Figure 5: 60% Static, 40% Dynamic Workload 30

Figure 6: CPU and Disk Utilization Under 60% Static, 40% Dynamic Workload 31

Figure 7: Performance Differences Between Nodes of Identical Hardware and Software 32

Figure 8: 60% Static, 40% Dynamic Using 2 Back-end Nodes 33

Figure 9: SMP vs. Uniprocessor nodes (4 CPU's total) 35

Figure 10: Scalability from 1 to 4 back-end nodes 36

List of Tables

Table 1: Static Workload 24

Table 2: Dynamic Workload 27

1 Introduction

The World Wide Web is increasingly being used as a basic infrastructure for a variety of Internet services [1] and its user pool is skyrocketing in numbers [2]. Network bandwidth that is available to clients is increasing at a higher rate than what servers based on single machines can support, and this will lead to server-side bottlenecks. A highly reliable, cost-effective web server system is a key solution to these demands. We have surveyed the literature in the area of high-performance web service, and conducted a performance study of a prototype Linux / Apache SMP cluster under a variety of web service workloads using the content-aware distribution policy *Client Aware Policy* (CAP). This study includes the difference in performance of uniprocessor versus SMP nodes in the context of web service, and the scalability measured by the addition of more nodes to the cluster.

1.1 Popular Website Workloads

As an example of what popular web sites are experiencing, Guinness World Records has recognized the official web site for the 1998 FIFA World Cup (www.france98.com) for setting four different Internet records [6]. The first is "Most Visited Web Site for an Event" as the site logged over 1.1 billion hits during the period between June 29 and July 12, 1998. The second record was for the most pages viewed in one minute, topping out at 235,356 on June 29, 1998. The third record is for the "most pages viewed in one hour" at over 10 million on June 30, and fourth is "most

pages viewed in 24 hours" at over 73 million (also on June 30, 1998). Guinness World Records has recognized Microsoft's www.msn.com collection of web sites as holding the record for "Most Day-to-Day Visits" when their servers logged 10.5 billion page views in March of 2000.

1.2 Motivation for High-Performance Web Service

Today's busy web sites must be highly available. As Internet usage develops and changes, there is a growing dependence on having such information readily available. In stark contrast to the early days of the Web, when the idea of the company web page was somewhat of a novelty, many Internet services, such as e-commerce, have become more and more "mission critical" because the penalty for system failures and service loss is greater than ever [1].

Research has shown that users are not willing to tolerate latency times greater than 8 to 10 seconds [4], and as traffic increases, the effective web server system must be scalable, keeping response times to a minimum. A system is scalable if the response time for individual requests is kept as small as theoretically possible when the number of simultaneous HTTP requests increases, while maintaining a low request drop rate and achieving a high peak request rate [5]. Response time is the length of time between the moment the client initiates the request and the time all of the information arrives at the client.

1.3 Web Workload Characterization

Web service workload has been found to be self-similar [7],[16] in nature, and has a heavy-tailed Pareto request distribution [17]. Self-similar with respect to web traffic means that the request patterns are "bursty", and periods of high and low traffic can span multiple time scales, from milliseconds to hours. The Pareto distribution is so-called heavy tailed in that (in the context of page request probability) smaller files are the most commonly requested, but as file size increases, the likelihood of larger pages being requested is a slowly decaying function (slower than a Gaussian distribution for example). With this distribution, a small percentage of the requests can use a disproportionately large amount of system resources. The need for highly available services requires that the web server system be able to fulfill the requests in an acceptable time frame during periods of peak usage, and these are often difficult to predict. Keeping a sufficient amount of resources available for effectively handling peak usage can be expensive and difficult to justify when those times of heavy demand are infrequent.

1.4 Web Server Architectures

Web server architectures currently in use include single uniprocessor machines, single SMP machines, uniprocessor or SMP clusters, and mainframes that can run many virtual servers simultaneously. In this thesis our focus is the use of clusters for web service. Clusters provide a high level of availability and performance, and are able

to scale with respect to throughput over time as demand increases. Another advantage is their high work/cost ratio, providing high throughput for a relatively modest price. There are subtle differences between one author's version of a cluster and another's, however, they often conform to a basic structure, that being a group of 2 or more commodity workstations connected together with high speed (100 Mbs or higher) interconnects. An example is shown in Figure 1.

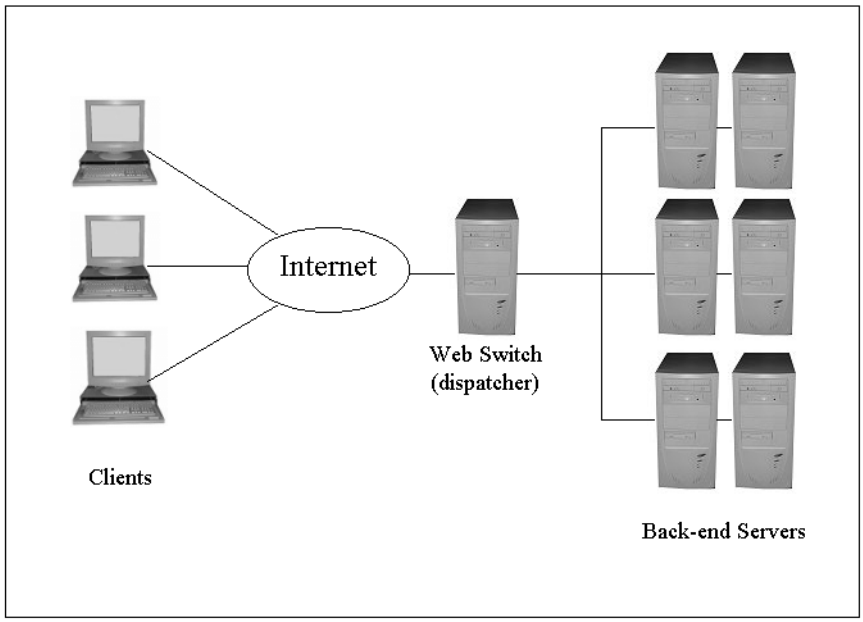

Figure 1: Example Web Cluster.
Clients issue requests to the same node (Web Switch) and these are then redirected to an appropriate back-end node.

These nodes of the cluster can be (but are not always) heterogeneous, and many researchers have designed their clusters to support the easy addition or removal of heterogeneous nodes as necessary based on demand. A common architecture places a

cluster of nodes behind a public point of contact (the head node, also referred to as the dispatcher, distributor, or the web switch). The clients' requests are distributed transparently, giving the appearance of a single machine. This type of solution is much less expensive than a mainframe or similar "high-end" system composed of a similar number of nodes.

There are many choices for distributing the document store across the nodes of the cluster. The document store is the collection of all resources that the web site can serve users, and can be shared among nodes either by a shared file system, such as the Network File System (NFS), or by replicating data across the nodes. Other ideas include analysis to precisely mirror the most popular content across all nodes, and distribute the rest by static partitioning. Static partitioning involves dividing up the document store so that a particular page or resource is located on a specific machine.

There are many different network topologies for web clusters. Often the cluster has a hierarchical structure, where the cluster maintains a public front-end node, and is connected to one or more private back-end nodes. This provides a single interface for all users to request services from the system. There are also multi-tier topologies where the head node may communicate with one or more mid-level nodes, who in turn communicate with back-end nodes.

1.5 Load Distribution and Balancing in Web Clusters

One of the many research areas in the design of web server clusters is that of load distribution. Load distribution solutions deal with balancing the workload among

all nodes of the cluster. Ideally, the nodes will be loaded such that they experience similar levels of CPU, disk and network utilization, keeping all nodes equally "busy". A popular approach to this distribution problem is to set up the cluster so that the public head node receives each request and directs it to the back-end nodes. Implementations have included Layer 4 switches (referring to the OSI protocol stack network layer), so that requests are directed to one of the back-end nodes via IP address modification according to some set of metrics, but do not include examination of the actual HTTP request [18]. These network level "content-blind" algorithms require server state information that must be regularly updated so the switch can make appropriate distribution choices, but are efficient and scale well provided they are tuned appropriately. Another approach is to use a Layer 7 switch, which is an application level implementation. This type of switch examines the HTTP request and forwards it to an appropriate back-end server. These algorithms have the advantage of being aware of what type of request is being made (e.g. static or dynamic), at the expense of extra system resources. For example, a simple request for a static page with a few inline graphic images has a relatively low demand on system resources, while other requests that involve transactions with a database management system or those needing to use public key encryption for secure transmission can create resource demands that cause orders of magnitude greater response times. This degree of control offered by Level-7 algorithms make it a viable alternative to the Layer 4 approach since the site operator can easily tailor the architecture to serve specific kinds of requests. One set of servers might handle only static content where much of the document store is cached

and can be returned quickly. Another set of servers may handle disk intensive content. If the nodes are heterogeneous, then the less powerful machines might be assigned to serve content that requires fewer system resources to deliver.

1.6 Load Distribution Algorithms

A variety of algorithms have been examined for web switch distribution. These are generally classified as either Layer-4 or Layer-7 as defined in the previous section.

Two examples of Layer-4 policies include random (RAND) which chooses a back-end node at random, and round-robin (RR) which always chooses the "next" back-end node in circular array fashion. The RR policy can be enhanced with the addition of server-state information to give each back-end node a "weight" relative to the server's current load to consider with the RR policy. This is referred to as weighted round-robin (WRR). These algorithms do not require knowledge of what content is being requested, and therefore can be handled at the network layer (IP address).

Two example Layer-7 algorithms are Locality Aware Request Distribution (LARD) and Client-Aware Policy (CAP) [8], also referred to as Multi-Class Round Robin (MC-RR) [9]. The LARD policy distributes requests based on which back-end node likely contains the page in its main memory cache, so the benefits of locality are realized. The CAP policy, developed by Casalicchio and Colajanni, distributes requests based on their expected impact on the back-end node's system resources.

Both of these algorithms require that the contents of the request be known to make the decision, hence the Layer-7 (application) categorization.

1.7 Contributions of this Research

Our contributions to this area of research include the comparison of CAP policy to RAND and RR policies. Previous work compared CAP to LARD and WRR, but did include RAND or RR. It is of interest to know how an algorithm behaves relative to RAND which is truly a baseline for comparison. We are also interested in the performance compared to RR, since many current implementations for request distribution and load balancing include this policy. These three algorithms: CAP, RR and RAND also share a commonality in that they are relatively simple to implement and do not require any back-end node state information, unlike LARD and WRR.

In comparing the CAP, RR and RAND algorithms, our research includes more detailed information regarding our workload content than many other published results. One major issue we encountered during this study is the lack of information to characterize typical workloads, including static/dynamic content probability ratios and response times for typical dynamic requests. In providing a more detailed view of our workloads, we hope that future research will have a better notion of realistically comparing published results, even if the workloads do not model a particular website.

Our work is incremental to those that investigated the use of Linux/Apache based web clusters, as our study uses the next generation versions of this software. The most recent published data we found included prototypes built with Linux 2.2 and

Apache 1.3. The nodes in our testbed are running the newer Linux version 2.4 kernel and Apache 2.0 server package utilizing the threaded multi-processing module *Worker* that handles connections with threads rather than processes.

Another contribution is our use of a prototype with dual-processor nodes. We are interested in the performance improvements realized by using SMP nodes for web service. Our study includes measurements of the cluster with back-end nodes that are dual-processor machines running the Linux-SMP kernel. Previous research in the area of content-aware request distribution where the web switch exists as a single layer in front of the back-end nodes included studies exclusively on uniprocessor architectures. Recently, a research group created a prototype on a Linux 2.4 cluster with dual processor nodes [24], but this was a multi-tier approach that differs significantly from our performance study.

1.8 Summary of Results

Our research shows the capacity of the reverse-proxy to redirect requests is approximately 1450 connections/sec. We show that CAP policy outperforms the RAND and RR policies by as much as 29% under the workloads consisting of mixtures of static and dynamic content, but no appreciable improvement is realized for workloads consisting of only static content. We demonstrate that for web service using the CAP policy under moderate workloads, the performance gain for dual processor SMP nodes is nearly 100% greater compared to using uniprocessor nodes, and the increased throughput by the addition of more nodes to the cluster is near linear. We did

observe that under very heavy workloads, the improvement in throughput for a dual-processor node versus a uniprocessor node is less than 100%.

1.9 Outline of this Paper

The general content of this paper is as follows: Section 2 discusses previous work in this area of research including the LARD and CAP policies and example web switch implementations. Section 3 describes our experimental design including the testbed architecture we used in our performance measurements and some discussion of software details. Section 4 discusses our experimental method and performance results, and we conclude in Section 5 including a discussion of potential future work for this area of research.

2 Related Work

Previous work in the area of content-aware request distribution policies has yielded two very different approaches to load balancing. The first is the policy known as Locality-Aware Request Distribution (LARD) [11], and the other is Client-Aware Policy (CAP) [8]. The performance of these policies (and others in this area) is evaluated in comparison to well-known algorithms such as random (RAND), round-robin (RR) and weighted round-robin (WRR). The LARD and WRR policies use state information from the back-end nodes for load balancing considerations, while RAND, RR and CAP do not. The random policy (RAND) chooses a back-end server at random from the available pool. The round-robin (RR) policy chooses a back-end with subsequent requests going to the "next" server in a circular array fashion.

2.1 Locality Aware Request Distribution Policy (LARD)

The LARD policy's strategy is locality-based, directing requests to the server that contains the information. The idea is to have the requested content readily available in the back-end servers' main memory caches, minimizing the response time delays caused by disk access where possible. LARD is not purely a locality based strategy. A purely locality based strategy is static partitioning, where the document store is divided among all of the back-end nodes. For example, if the cluster is comprised of 10 back-end nodes, then each node will contain approximately 1/10th of the document store. The difference in LARD compared to static partitioning is the inclusion of server state

information. There is a back-end node threshold load value based on some set of metrics. The metric used by the LARD algorithm is the back-end node's number of open connections. When the web switch receives the request, an attempt is made to direct it to a back-end node based on static partitioning, but if the load value exceeds the threshold level, the switch will direct it to a less loaded node, if any exists. Choosing when to redirect to an alternate node versus experiencing a small load imbalance is decided by tuning parameters that are part of the LARD algorithm: the high and low connection threshold values. The algorithm can be augmented so that highly popular pages can be replicated across all nodes, minimizing the chance that one node will become overloaded due to repeated requests of the same page, but replicating many pages across all nodes defeats the general purpose of maintaining locality.

The amount of state information required by LARD is minimal: the number of open connections. No information such as CPU and/or disk utilization is needed for the LARD policy. One shortfall of this algorithm is that efficient locality is difficult to maintain for highly dynamic content in the respect that caching such content can be expensive or impossible. Web traffic is becoming increasingly dynamic, which suggests that the effectiveness of a distribution policy based on the LARD algorithm is limited. The performance of LARD is covered in Section 2.5.

2.2 Client Aware Policy (CAP)

The other major content aware distribution algorithm, Client Aware Policy (CAP), distributes requests based on the category of the content (e.g. static, dynamic,

secure, etc.). The decision of which back-end will receive the request is determined by the expected impact on system resources. This policy has also been referred to as Multi-Class Round Robin (MC-RR) [9]. Like LARD, it is a Layer-7 policy that makes decisions based on the content of the actual HTTP request, but the idea is to keep requests of each category distributed evenly among the back-end nodes so that each server is handling a similar number and variety of workload categories. Unlike the LARD policy that maintains a count of open connections per back-end, CAP does not require any server state information. This makes the implementation simpler than that for LARD. Previous research that studied the performance of this policy is discussed in Section 2.5.

2.3 Web Content Workloads

We can categorize existing web sites into three broad categories [9]: *Web Publishing*, *Web Transaction* and *Web Multimedia* sites. *Web publishing* sites contain primarily static and lightly dynamic content, such as static HTML pages and dynamic requests that do not make intensive use of resources. Examples include a static page with some inline graphics, or a small CGI process that returns a counter with the number of page hits. These services are primarily CPU bound. *Web transaction* sites provide dynamic content that requires more complex database queries and other system resources, and possibly requires secure, encrypted transmission of sensitive data. An example is a web site that allows personal banking customers to get account

information, transfer funds, etc. These sites provide services that are CPU and/or disk bound. *Web multimedia* sites include content such as streaming audio and video using specialized hardware and software. Our study does not attempt to consider this last type of web workload. For the purposes of this study, we use a synthetic workload consisting of static pages, light CPU intensive dynamic content, heavy CPU intensive content, and a resource representing a combination of CPU and disk intensive content. *Web publishing* sites contain both static and dynamic content, but the ratio is more heavily weighted to static content. A *Web transaction* site, in comparison includes a higher percentage of dynamic content.

2.4 Web Switches

A web switch node has the duty of relaying the requests to an appropriate back-end server, and there are a variety of methods to accomplish this. Three of them include TCP splicing, TCP hand-off and Reverse-Proxy forwarding.

2.4.1 TCP Splicing

In the TCP splicing approach [12], the web switch receives the connection from the client, examines the HTTP header information, then makes a decision to forward the TCP packets to an appropriate server. At this point, the web switch is masquerading as the client. The response from the back-end server returns through the web switch and is forwarded to the client. As long as the connection exists, the packets travel up through to the network layer of the protocol stack before they are forwarded

to the appropriate end (similar to network address translation, or NAT at this point), hence the term "spliced connection." This requires modification of the kernel in the web switch node, but no changes to the back-end nodes. All traffic between endpoints travels through the web switch.

2.4.2 TCP Handoff

The TCP handoff mechanism [11] is an improvement on the TCP splicing approach, where all the incoming packets from clients pass through the web switch, but after passing them on to the appropriate back-end node, the responses are sent back using a network connection that does not pass through the web switch, in effect the connection is "handed off" to the server so that the reply returns to the client directly. This has been shown to be a notable performance improvement over TCP splicing. This type of switch requires kernel modification of both the switch and the back-end nodes.

2.4.3 Reverse-Proxy Forwarding

The use of Apache as a reverse-proxy web switch was described by Ralf Engelschall [14] and used in development of the CAP policy [8]. This is an adaptation of the Apache web server software [15]. In addition to the "core" modules, Apache has a module called mod_rewrite that can rewrite the URL string, choosing and replacing any portion of it with a series of regular expression type rules that the webmaster can create in a configuration file. The other notable module is mod_proxy which provides the request forwarding functionality. The request's URL string is modified in

mod_rewrite, replacing the host portion of the URL with that of a back-end node. The request is forwarded to that back-end node via the proxy mechanism. This approach requires no kernel modification to implement, however, being a user level application, it has a higher overhead than kernel level implementations like TCP splicing or handoff. Research has shown that switches operating at Layer-7 pose scalability problems [13] above ten back-end nodes, but for the purpose of studying dispatching policies on a small cluster, the reverse-proxy is sufficient.

2.5 CAP and LARD Performance

The CAP authors examined the performance of their algorithm in both simulation and prototype trials [8]. In the simulation experiments, LARD outperformed CAP and WRR for the static workload due to its effective exploitation of locality. For light dynamic requests the performance of CAP and LARD was about the same, both giving better results than WRR. For heavy dynamic requests, such as in Web Transaction sites, CAP clearly outperformed both LARD and WRR.

In the prototype trials, the authors used a reverse proxy web switch based on the Apache software, and found that while CAP and LARD performance for static content was about the same, CAP was more effective than LARD for both light and intensive dynamic workloads. Their conclusion is that LARD is appropriate for web publishing sites that use mostly static and some light dynamic content, but for modern web transaction and commerce sites, a policy like CAP will yield better results. The authors

make another point that CAP is a robust policy in that it does not require any special tuning parameters, unlike the LARD and WRR policies. If not tuned properly, these latter two can yield results worse than that for random distribution.

Many authors have noted the potential bottlenecks in distributing requests at the application layer and have proposed alternate architectures. One group who focused on the implementation of the web switch rather than distribution algorithms has determined that it depends on the particular website whether to use content based routing at the web switch or to use TCP routing (a Level 4 switch) [21]. In the case where the back-end nodes are likely to be the bottleneck, they suggest using content-based routing. If the content-based switch is likely to be the bottleneck, then use TCP routing (such as the splicing or handoff techniques described in Section 2.4). They discuss the performance benefits of implementing these techniques in an embedded operating system, yielding higher performance than that of a general purpose O/S.

Another group suggests a strategy known as WARD [22]. The WARD policy partitions the working set into a small, frequently requested group called *core*, which is distributed across all nodes, and the group of less requested pages is partitioned similar to static partitioning. Their algorithm includes *ward-analysis* which computes the optimal *core* size. It takes into account access patterns and cluster hardware characteristics. Their studies from simulation results describe a large improvement over both RR and static partitioning strategies. It has not yet been implemented on a prototype architecture

3 Experimental Design

The main goal of this study is to show that a content-aware policy such as CAP on a prototype Linux / Apache architecture using SMP nodes provides a substantial improvement over random and round-robin distribution for workloads that contain dynamic content. A second goal is to demonstrate the utility of SMP nodes for web service. We do this by performing measurements on our prototype cluster using the CAP, RR and RAND algorithms, and measure the performance differences between dual-processor SMP and uniprocessor nodes. We used the latest available versions of both Linux and Apache to take advantage of their reported increases in performance over previous releases. A third goal is to study the scalability of the cluster, measuring the performance as more nodes are added.

3.1 Testbed Architecture

Our testbed for this study is a subset of a forty-eight (48) node Intel Pentium-III based cluster ("Wyeast") located in the Department of Computer Science's High Performance Computing Lab at Portland State University. This subset consists of seven (7) identical machines with the following specifications:

- Dual (2) 866 MHz Intel Pentium III processors on an ECS D6VAA dual-socket mainboard
- 512 MB PC133 SDRAM
- 20 GB Hard Disk

- 3Com 3c905c 10/100 Network Interface
- Linux O/S, kernel 2.4 (RedHat 7.2 distribution)

These nodes are all connected via a Cisco 3548 switch, providing 100 Mbs bandwidth between nodes.

3.2 Testbed Software Details

We used the recommended Linux 2.4 kernel tuning parameters as listed for the SPECweb99 benchmark [23]. The tuning parameters modify the size of socket input and output queues, range of allowable ports, and allowable simultaneous TIME_WAIT sockets. In our tests, these parameters did not have a noticeable effect on our results, suggesting we were not stressing the system in the areas that needed these parameters changed from the default values.

This version of Linux utilizes a new file system, called EXT3, which is a journaling file system now included with RedHat distributions. The journaling component adds transaction-based integrity to the disk, but does not sacrifice performance. A paper describing the early stages of this development is available for more information [20].

To guarantee dedicated use of the network, each testbed client and server is located on an interior, private node of the local area network. One of the machines is designated as the web switch, and runs a thin build of Apache 1.3.24 as a reverse proxy. The RAND policy is already available in the Apache source code distribution. We

modified the source code to support the RR and CAP policies. Four of the machines are designated servers that each run a general purpose build of Apache 2.0.28. The last two machines are designated clients that have the job of sending requests to the web switch. One of these two has the Webstone [10] version 2.5 package installed, which during test runs, executes the *webmaster* module, which in turn calls the *webclient* module. The *webmaster* process starts a number of *webclients* on both the local machine and any other designated client nodes. We created workload profiles that Webstone uses to generate requests.

We did not modify the Webstone executable in any way other than applying a patch to make it compatible with the Linux operating system. The webclient modules request pages without any delays. Some researchers have modified the program to include user "think time", or to make the request arrival times at the server "bursty". Adding think time would make the request profile more authentic, more closely modeling real client behavior, but for the purpose of comparing algorithms without regard to absolute capacity of our system in "real client" numbers, we feel the lack of think time does not prevent the study from producing useful information. One way to look at this is that Webstone is sending the requests at a rate where the arrival rate is as high as possible during a burst. This should make the reported throughput results worse, not better. A more uniform request rate should also increase the reproducibility of the results. If think time been implemented, we would expect to see a higher number of client processes to yield the same throughput. Our study uses the stock version 2.5 code without modification.

Each Apache build, except the web switch reverse proxy, utilizes the new threaded Multi-Processing Module (MPM) *Worker*. The previous major release of Apache (1.3) uses a pre-forking model where upon starting the webserver, the master process forks off a number of child processes that each handle connections. If the number of connection requests exceed the capacity of the available processes, more child processes are spawned according to criteria in the configuration file (usually constrained by available RAM). By contrast, the newer 2.0 version is threaded. The *Worker* module is designed so that connections are handled by threads rather than processes. Launching the webserver software spawns a number of child processes, and each child process then spawns a set number of threads, each of which handles connections, realizing the benefits of lower system overhead compared to the non-threaded pre-forking model. The number of threads is increased to match the demands of increasing connection requests.

In early experiments we observed with an Apache 2.0.28-based reverse-proxy web switch that the switch became overloaded much more quickly than we expected. We then tried the previous Apache version, and had much better results. Some modules were completely rewritten for version 2.0, and this may explain the performance drop.

To summarize the portions of our study that included custom or modified software, they include: creation of workload dataset files used by Webstone (called "filelists") for request generation (multiple profiles including mixtures of static and dynamic content); creation of CPU and disk intensive dynamic service modules (used

by Apache), that represent dynamic requests; modification of the mod_rewrite module in Apache 1.3 to implement the Round Robin and CAP algorithms; building a specialized reverse-proxy version of Apache 1.3 using a customized run-time configuration file and back-end node distribution maps; writing small scripts to aid in reducing data and to gather CPU and disk utilization information, and the application of Linux 2.4 kernel tuning parameters to all back-end nodes of the cluster.

4 Experimental results

In this section we detail our results for the performance of CAP, RAND and RR on workloads consisting of various mixtures of static and dynamic content. We then describe the performance benefits of using SMP versus uniprocessor nodes, and the measured increase in performance as more nodes are added to the cluster.

We chose to measure performance using the metric *connections per second*. The literature shows this is a common metric used by many researchers in the area of cluster based web service. We also could have chosen network bandwidth throughput as megabits per second, or request response time. We did not choose network bandwidth as this does not accurately reflect the server throughput when dynamic requests make up a significant portion of the content. While static file response times are relative to their file size, dynamic requests can take orders of magnitude longer to service, yet the amount of data sent through the network with the result can be much smaller, and so have no such relationship to their response time. We did not use response time since the stock Webstone software only provides 1) aggregate minimum, maximum and mean response times for all requests, and 2) minimum, maximum and mean response times for individual requests. We made no modifications to the software. Webstone could be modified to generate more useful aggregate response time information, such as "number of requests with a response time less than or equal to X", however in the scope of this study, we are satisfied with the information provided by connections per second.

4.1 Capacity of the Reverse-Proxy Web Switch

To determine the capacity of the web switch to forward requests to the back-end nodes, we configured Webstone to request pages of zero size. This way we could measure the performance of the switch with a minimum amount of data transfer, eliminating our 100 Mbs bandwidth as a potential bottleneck.

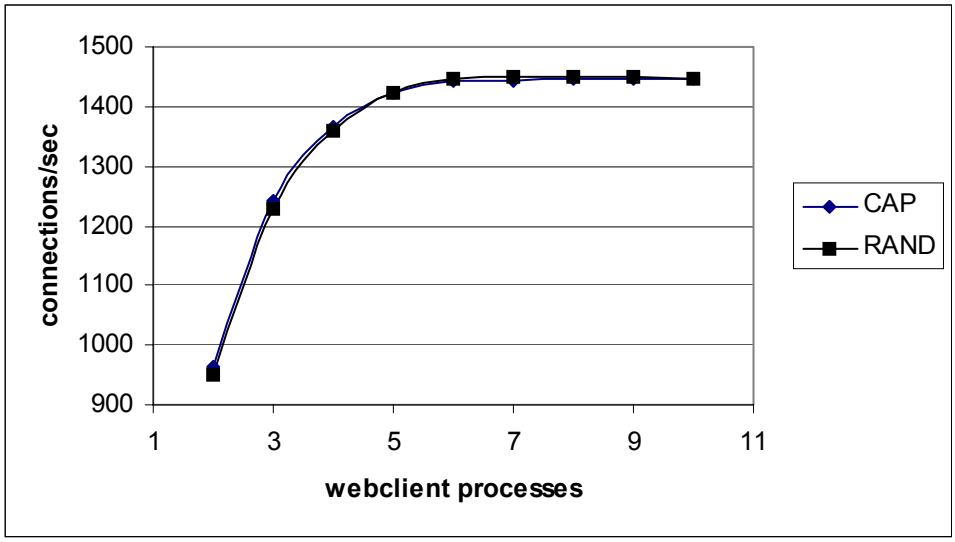

Figure 2: Capacity of the reverse-proxy web switch.
As the number of *webclient* processes increases, the capacity of the webswitch reaches a maximum of approx. 1450 connections/sec.

The results, shown in Figure 1, indicate the maximum throughput at approximately 1450 requests per second.

It should be noted that each Webstone client process (*webclient*) corresponds to a process running on the designated client node where it issues a request, waits for the result, and after receiving the response issues the next request. Unlike real human users, there is no "think time" between requests. Therefore, the requests generated by

Webstone do not represent a number of real users, but are useful in applying loads of varying intensity to the web server. Additionally, this may apply to cases where clients are not humans at all, but rather computer systems transacting with servers using the HTTP interface. We use the throughput metric connections/second, as this gives the best indication to how well the cluster is servicing requests, especially with dynamic content where size of the returned message(s) is not necessarily proportional to the response time.

The switch was tested using both the RAND and CAP algorithms, and the results show that they perform essentially the same. We expected that the CAP policy might show slightly lower performance compared to RAND since there is contention for a lock on the shared memory array containing the index of the "next" back-end node. The results show that the overhead is insignificant.

4.2 Static Workloads

We tested the CAP, RR and RAND policies using a static workload. For this test we used a static fileset included with Webstone. These files do not contain HTML formatting, but Webstone does not differentiate between files formatted as HTML or plain text. Each file does have the .html extension. Apache is configured to recognize the .html extension as an HTML formatted file and returns it with the appropriate content headers. The static fileset consists of a set of text documents from 500 bytes to 5 MB, and is configured with the following probability distribution shown in Table 1.

File size	p(x)	Avg Response Time
500 bytes	0.35	2 msec
5 KB	0.50	3 msec
50 KB	0.14	7 msec
500 KB	0.009	47 msec
5 MB	0.001	500 msec

Table 1: Static Workload
Static files of the various sizes used, the probability that a file of that size will be requested and the average response time on a server under light load.

Early experiments showed that the five files in the static workload were being cached, showing zero disk utilization under heavy load. To more accurately simulate a static workload of a large working set, we copied the files using unique filenames, keeping the distribution of file sizes the same. Our modified static working set consists of 2000 files, totalling about 38 megabytes in size. The average response time for all files was 4 msec for the static workload.

If all the files were considered to be the same resource category, the CAP distribution would be essentially round robin, so the largest two files, 5 MB and 500 KB, were placed in one category, and the 500, 5KB and 50KB files in another. The performance data was collected using three trials (Webstone "runs") of 10 minutes each. The variance shows good agreement between trials, and so averaged results from the three trials are presented. The results for the static workload using RAND, RR and CAP are shown in Figure 3.

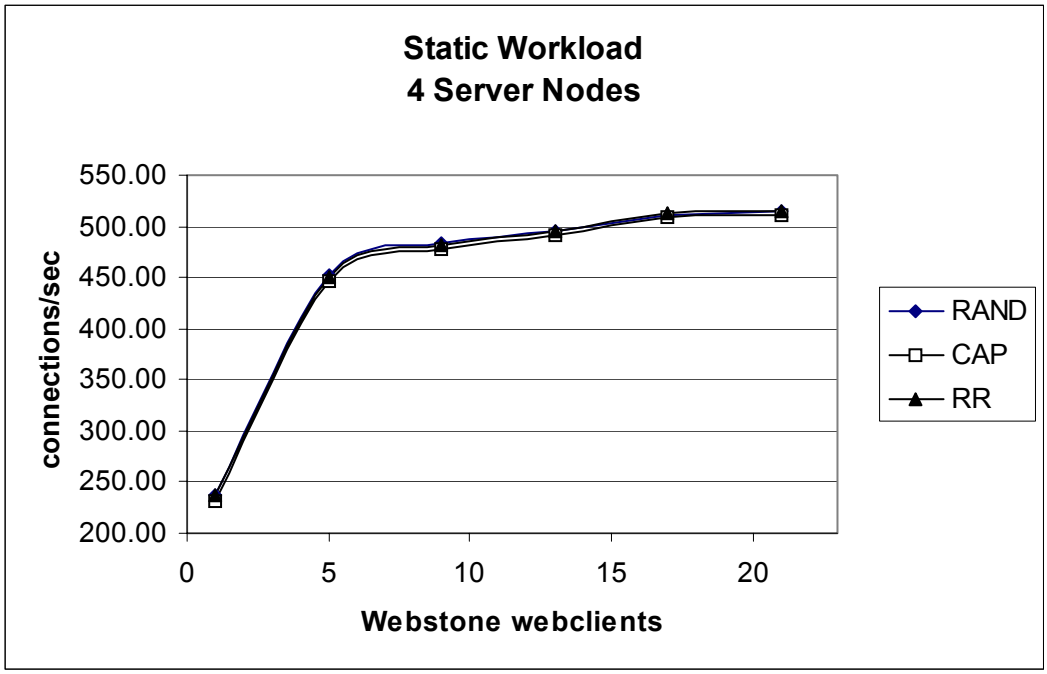

Figure 3: Static Workload Results
All three policies have similar performance for static workload within the bounds of the 100 Mbs network bandwidth.

Figure 3 shows that there is no advantage of CAP over RR or the RAND distribution on our testbed. This is consistent with previous research. In fact, all algorithms are constrained by available bandwidth, as the server throughput was measured at 75-80 Mbs for the trial with 10 client processes. This does not include the extra bandwidth used by the reverse proxy in relaying each request, as each request creates two connections: one from the client to the web switch, and one from the web switch to the back-end. CPU and disk utilization was sampled during a separate trial with 10 client processes showing 5-10% CPU and less than 1% disk utilization on each

node. For static workload, the system could provide more throughput, but the bandwidth prevents fully realizing this capability.

4.3 Dynamic Workloads

We tested the system using workloads that included dynamic content. The performance data was collected using three trials of five minutes each, and we present the averaged results. The CAP algorithm should provide better performance for heterogeneous services, so workloads were synthesized to include multiple classes of requests. To simulate dynamic workload, three CGI programs were created: one representing CPU lightly intensive dynamic content, one for CPU heavily intensive dynamic content, and one for disk bound dynamic content. These programs were created to simulate dynamic workloads on the web server, but do not attempt to model a particular application. We started with the basic notion of dynamic content requiring an order of magnitude more processing time than static content. This is represented by the *CPU light* service. This program consists of a simple loop that makes math library calls. The *CPU heavy* service requires twice the processing time, done by increasing the number of loop iterations. The *Disk* service requires a similar amount of time as the *CPU heavy* service, but instead of looping through math library calls, it reads a text file, writes it back to disk, and finally deletes the newly created file. The file handle is opened using the O_SYNC flag, causing the process to block until the contents are actually written to disk. Table 2 lists these services with their average response times on a server under low load. These probabilities were chosen based on work by

Casalicchio and Colajanni, but since no information is available as to their response times for a particular resource, this cannot be considered the same dynamic workload that they used for their experiments.

Service	p(x)	Avg Response Time
CPU light	0.5	40 msec
CPU heavy	0.3	100 msec
Disk	0.2	100 msec

Table 2: Dynamic Workload
Shown are the probabilities for a particular resource when dynamic content is requested, and the average response time on server under light load.

These were written in C to use the CGI interface, compiled and placed in the appropriate CGI directories on the back-end nodes. There was not a large concern for writing them this way, they could have been written using another language such as Perl just as easily. Recall from Section 4.2 that the static workload had an average response time of 4 milliseconds as a whole. The average response time for the *CPU light* service was written to have a response time of approximately 40 msec, an order of magnitude greater than the average static workload response time on a server under low load. The *CPU heavy* service has an average response time of 100 msec, or 2.5 times greater than the *CPU light* service. The *Disk* service has an average response time of 100 msec, but includes a high degree of disk activity, so it stresses the system in

a different way than the former two. These service programs, along with the static workload were combined to make two different mixtures: 80% static + 20% dynamic, and 60% static + 40% dynamic.

4.3.1 80% Static, 20% Dynamic Workload

A mix of 80% static and 20% dynamic workload was created using the static profile discussed in Section 4.2 with the addition of 10% *CPU light*, 6% *CPU heavy* and 4% *Disk* services. This example tries to approximate a *Web Publishing* site. The workload is a reasonable mix of resources requested according to previous research [8], but does not attempt to precisely model a particular website.The purpose here is to show effects of distribution algorithms on throughput of systems that offer heterogeneous services. The performance data was collected using three trials of 5 minutes each (we did not see a significant change in performance measurements between trials of 10 minutes and those of 5 minutes). The variance shows good agreement between trials, and averaged results are presented. The results of this workload are shown in Figure 4.

RR performed similarly to RAND, but CAP clearly performed better than both of these, but with increasing number of *webclients*, the benefits of CAP are lost, and the performance approximates that of RAND and RR. The improvement of CAP vs. RAND is approximately 20% at the 10 *webclient* process level. CAP performed 16% better than RR at this point. The performance is essentially the same at 25 *webclients*.

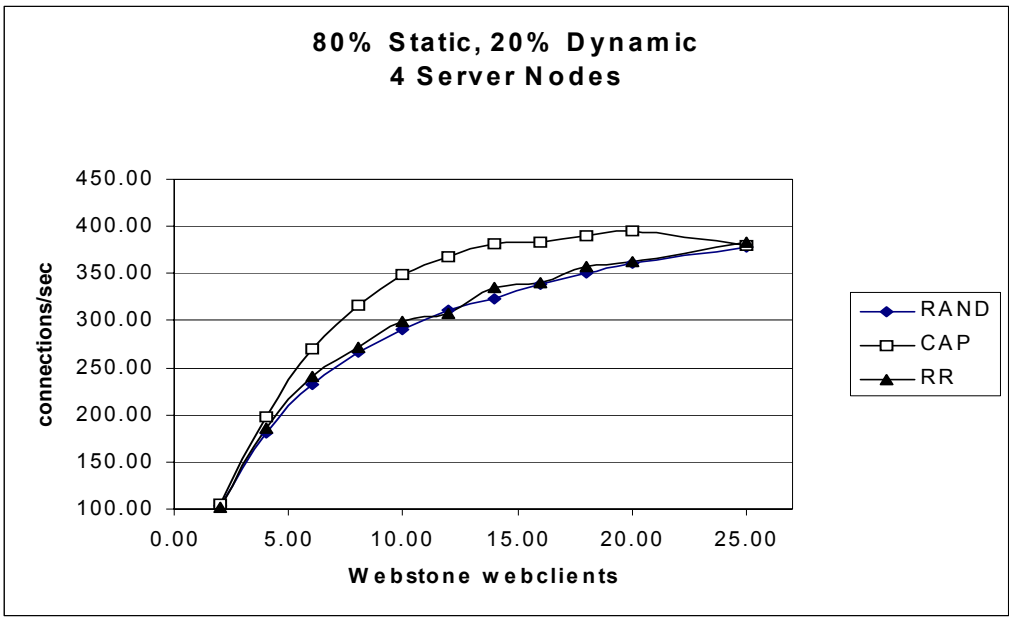

Figure 4: 80% Static, 20% Dynamic Workload
The benefits of CAP are realized with higher throughput when workloads include dynamic content, but degrades to the level of RAND and RR with an increasing rate of client requests.

4.3.2 60% Static, 40% Dynamic Workload

We increased the dynamic portion of the same workload file set from Section 4.3.1, creating a 60/40 mix, or 60% static and 40% dynamic content. This might represent a *Web Transaction* type site, with even heavier demands on system resources and longer response times than the previous workload. The performance data was collected using three trials of five minutes each. The results are shown in Figure 5.

As in the previous workload, RR performs somewhat better than RAND, but CAP performs better than either. The improvement at 8 client processes is 29% (at 10 processes it is 27% percent higher than RAND). Beyond 10 *webclient* processes, the server performance levels out. This data combined with the observations of the static and 80/20 static-dynamic mix suggests that the more dynamic the workload, the more CAP improves when compared to RAND or RR, but only to a certain point. If the server is heavily loaded, the performance degrades to the level of both RAND and RR.

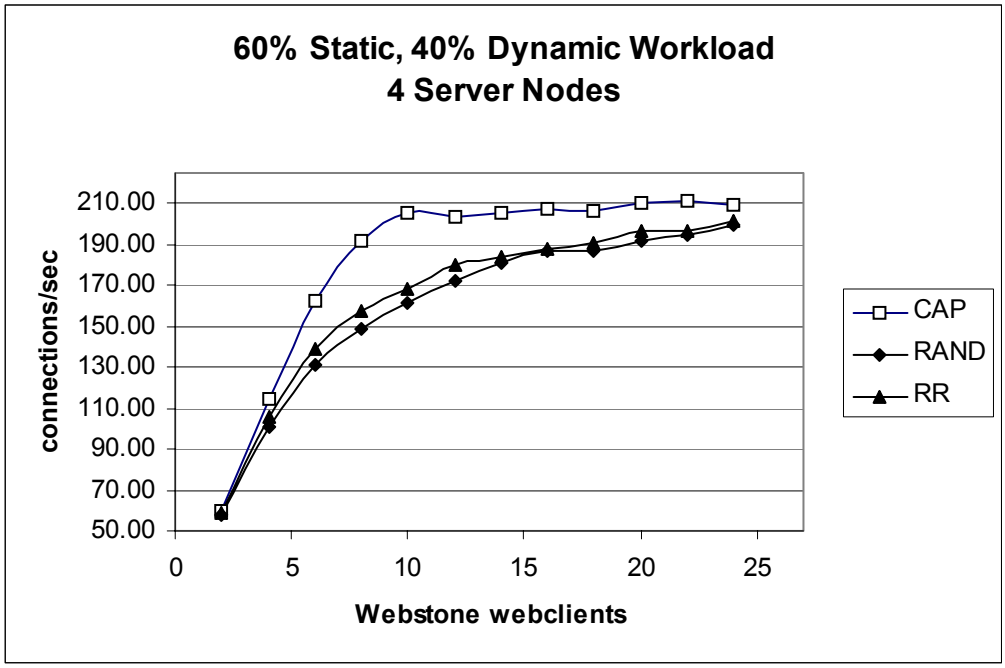

Figure 5: 60% Static, 40% Dynamic Workload
Increased level of dynamic content produces an even greater difference in performance between CAP and the other two algorithms at the 10 *webclient* level compared to the 80% static, 20% dynamic mix. With increased number of *webclients*, the performance approximates that of RAND and RR.

To investigate the reason behind the relative performance drop of CAP vs. RAND and RR, we examined CPU and disk utilization of CAP using the 60/40 dynamic workload with 10 *webclients* and with 14 *webclients*. These two points represent the load in which the server has essentially the highest throughput (using 10 *webclients*) and remains essentially the same with an increased number of *webclients* (14). Using iostat, we collected CPU and disk utilization at 15 second intervals on one of the back-end nodes while running Webstone with the 60/40 workload. The results are shown in Figure 6. The results show that utilization of CPU and disk resources is variable and does not lend insight to the reason behind the throughput increase being essentially flat between 10 and 14 *webclients*. Network bandwidth used for these workloads is below 20 Mbs, and so should not be an issue.

An additional test was run to verify that all back-end nodes are providing similar performance. The CAP algorithm makes the assumption that a particular request will have the same impact on system resources on every back-end node, provided they are built to be identical nodes (same hardware and software). We set the web switch node to direct requests to a single back-end node, and measured the throughput for each. The results are shown in Figure 7.

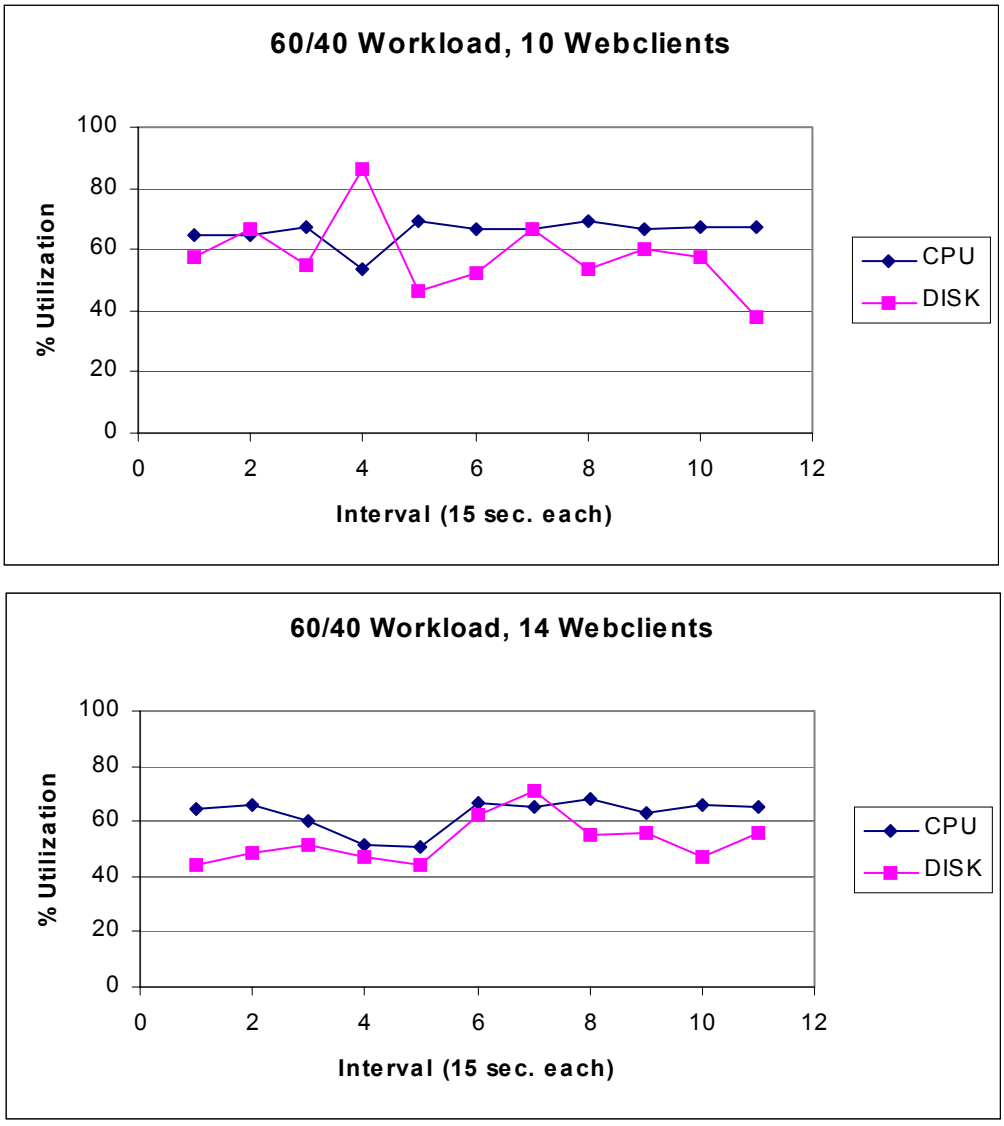

Figure 6: CPU and Disk Utilization Under 60% Static, 40% Dynamic Workload
Utilization is variable and does not explain the flat throughput increase when the number of *webclients* is increased.

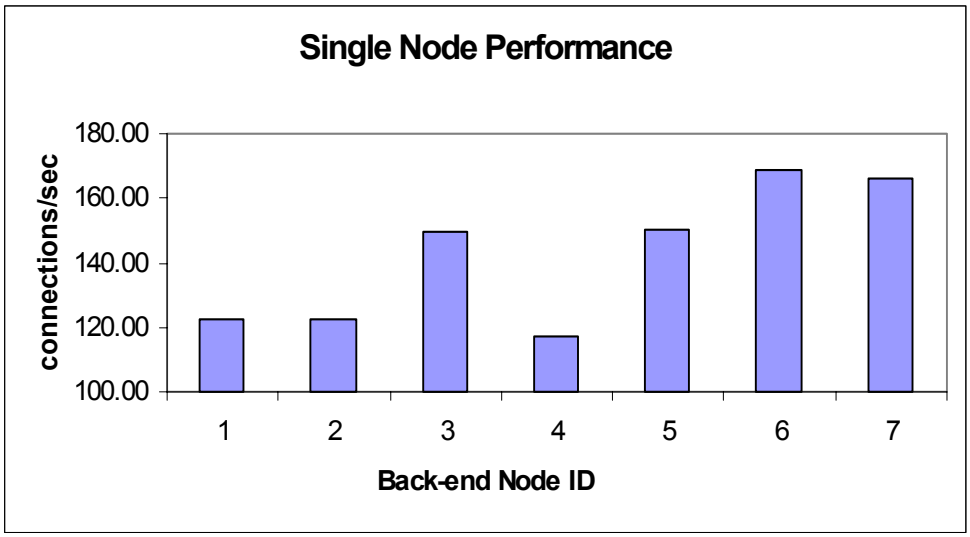

Figure 7: Performance Differences Between Nodes of Identical Hardware and Software
Nodes consisting of the same mainboard, drives, network cards and software show notable performance differences.

Figure 7 shows that even nodes built to be identical still can exhibit differences in performance. We used the most closely matched nodes (3, 5, 6 & 7) as the back-end servers. Note that the effects of varying performance levels should make CAP worse relative to RAND and RR, since CAP distributes based on the *expected* impact on resources. Nodes that are very closely matched should yield the best performance. Since the operating system on these nodes has gone through more than one upgrade since the cluster was built, it may be that a clean install would even out the performance. The reader should be aware of this possibility in performance difference among nodes if they decide to implement CAP on their system, since it would be a good idea to determine individual node performance to aid in selecting which nodes

will serve which class of requests, or troubleshooting an unexpected performance drop when one node has a significant performance difference from the rest of the pool. While the performance differences were unexpected, they point to one of the advantages of the CAP algorithm over RAND and RR for clusters in practice: CAP can compensate and balance heterogeneous nodes, as long as the set of nodes that will service a particular class of request is homogeneous. RAND and RR, which do not examine the content of the request, cannot make these server load-related decisions.

We observed that the performance increase of CAP compared to the other algorithms is dependent on a certain range in number of *webclient* processes issuing requests. Recall that the maximum performance increase for the 60/40 workload was at 8-10 *webclients* for 4 back-end nodes.

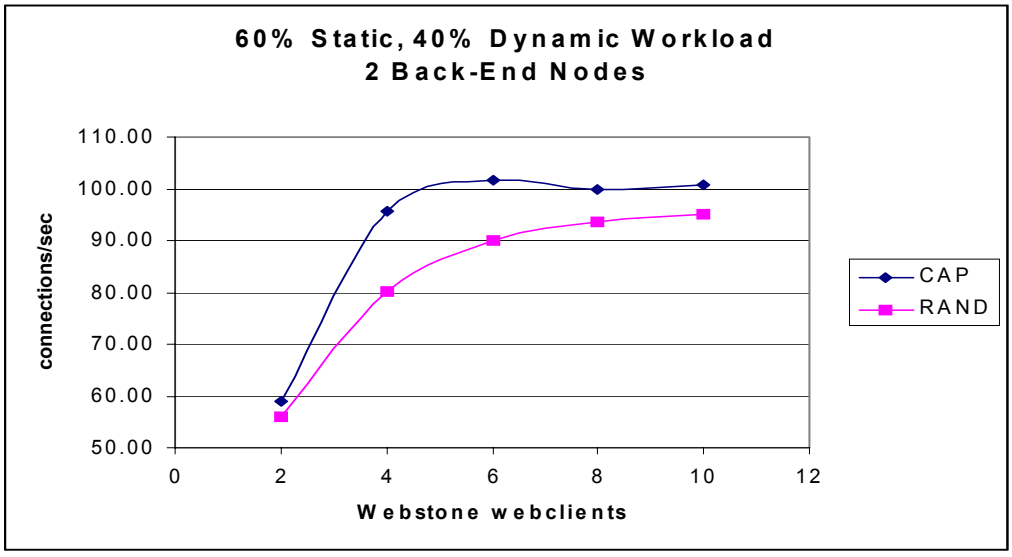

Figure 8: 60% Static, 40% Dynamic Using 2 Back-end Nodes. This figure shows how the maximum CAP improvement is seen with a *webclient* to back-end node ratio of approximately 5:2.

We ran an experiment using the CAP and RAND algorithms with 2 back-end nodes to examine this behavior (three trials of 5 minutes each). The results are shown in Figure 8.

This demonstrates that CAP's improvements in performance are sensitive to the overall server load, although we cannot immediately apply these numbers to a real scenario. It would require testing with an actual website to determine the appropriate number of back-ends to realize the greatest improvement.

4.4 SMP versus Uniprocessor Nodes

We are interested in the utility of SMP nodes in web clusters. To investigate the performance of SMP versus uniprocessor nodes, we conducted a set of experiments, contrasting the two modes of operation. The first test was to determine how much speedup was produced at the web switch by running in Linux SMP mode vs. UP (uniprocessor) mode. Running the same workload that was used for the switch capacity test (zero size file), we measured about 800 connections/sec. This is about 55% of the 1450 connections/sec that we measured while running in SMP mode.

We studied the effects of SMP nodes on the back-end servers running a static/ dynamic workload. We had four nodes available for the back-ends, so we ran 4 processors in SMP mode (2 x 2) vs. 4 processors in UP mode (4 x 1). For the SMP test, we used two of the back-end servers on the 60% static, 40% dynamic workload. For the UP test, we rebooted the back-ends to run the uniprocessor version of the kernel,

and ran the same workload using all four back-end nodes. Data was collected for three trials of 5 minutes each, averaging the results. The results are shown in Figure 9.

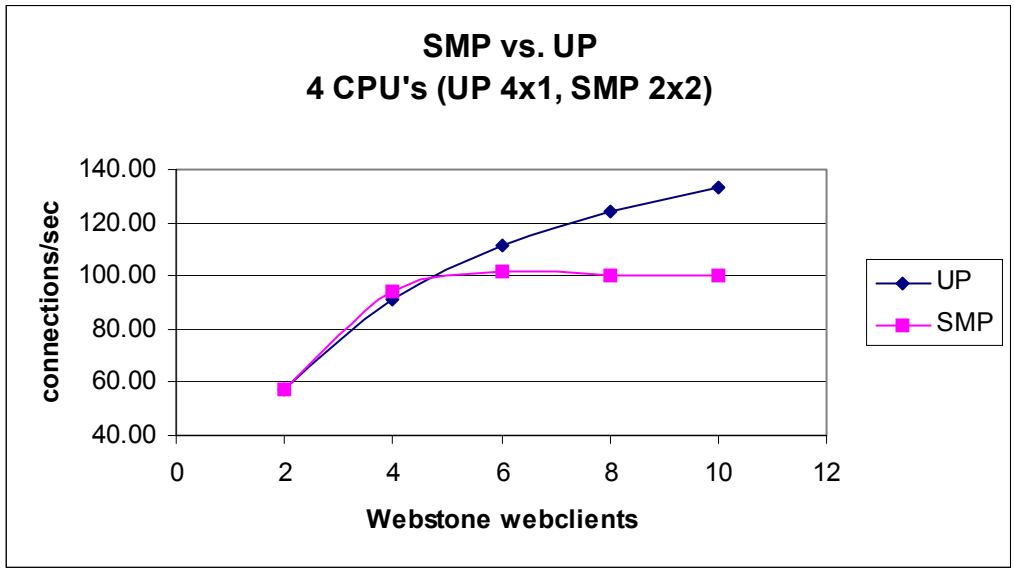

Figure 9: SMP vs. Uniprocessor nodes (4 CPU's total)
Although equivalent under lighter loads, 4 uniprocessor nodes show a higher throughput than 2 dual-processor SMP nodes under heavy workloads.

The SMP results show a similar throughput compared to the UP runs up through 4 *webclient* processes. After that, the UP runs show a greater throughput. Unlike the SMP system, the uniprocessor system does not have memory or I/O bus contention between processors. This may explain the higher throughput with the UP 4-processor system in this area.

4.5 Scalability

We examined the scalability of a system composed of 1, 2 and 4 back-end nodes. For this experiment, we used a 90% static / 10% dynamic workload running trials in SMP mode. The percentage of dynamic content was only 10% of the whole, but the dynamic services themselves were more CPU and disk intensive than those in previously discussed workloads, hence the "low" connections/second values. The results are shown in Figure 10.

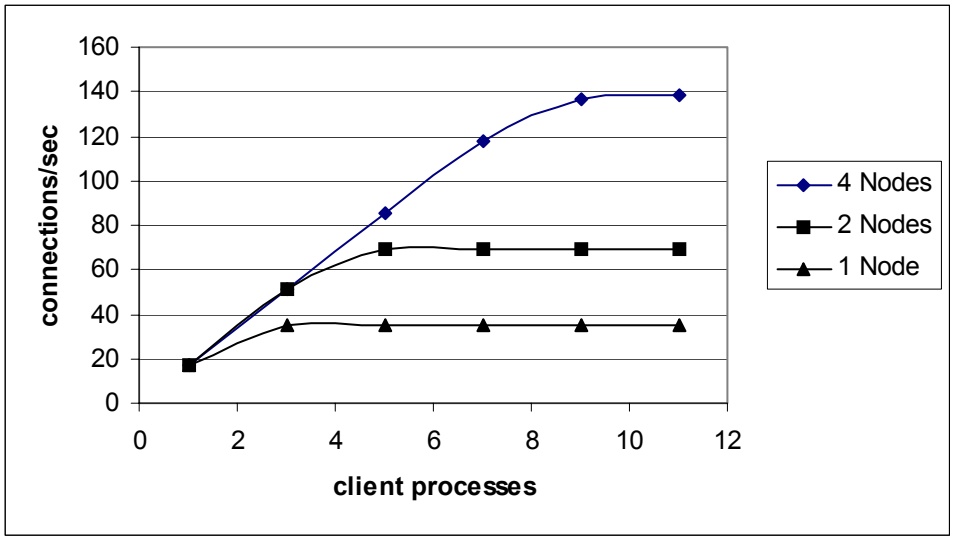

Figure 10: Scalability from 1 to 4 back-end nodes
The measured maximum throughput as more nodes are added to the cluster is nearly linear.

At workloads that result in maximum throughput levels, the results show a good agreement to the theoretical optimum of realizing nearly twice as much throughput when you double the number of back-end nodes. The trial using 11 client

processes measured the 4-node system at 138 connections/sec, which is 100% higher than the measured value of 69 connections/sec for 2 nodes, and 390% greater than the throughput of 35 connections/sec with 1 node.

Our experiments have shown that the performance of CAP is higher than that of RAND and RR on workloads containing some degree of dynamic content under light to moderate workloads. As the system becomes heavily loaded, the performance of CAP approaches that of RR and RAND. Workloads of only static content did not show appreciable differences, however our experiment was limited by available bandwidth. We demonstrated the performance of using SMP versus uniprocessor nodes where the number of CPU's is constant. The two dual-processor SMP nodes showed similar performance for lighter workloads, but showed poorer performance compared to the four uniprocessors under heavy load. The scalability of the cluster at maximum throughput was shown to be near linear in the performance improvements realized as additional nodes are added to the cluster.

5 Conclusions and Future Work

We have examined the content-aware distribution algorithm CAP and its performance compared to content-blind algorithms such as random and round-robin on a Linux cluster with SMP nodes. Our results do not completely agree with previous research of the advantages of CAP policy. While the performance of CAP exceeds that of RAND and RR under lighter workloads, we observed a reduced advantage of CAP over RR and RAND when the system is under heavy load.

The performance gains of CAP over RAND and RR improve with an increased concentration of dynamic workload, so web sites with highly heterogeneous content should perform better provided there are a sufficient number of back-end nodes. The algorithm is relatively simple to implement, and combined with its lack of any special tuning parameters other than content categorization, it is a choice to be considered for implementation in a web request distributor.

The performance of CAP, according to the most recent information available at the time of this writing, has not been examined on a Linux 2.4 cluster with SMP nodes where the request distribution is facilitated by a single web switch in front of the back-end nodes. We have shown in our experiments that the performance of a 2-way SMP node is equivalent to two uniprocessor nodes until the system enters a heavily loaded state. Even considering this effect, the performance increase from uniprocessor to SMP, combined with the relatively low cost of the additional processor and supporting mainboard suggest it is a worthwhile upgrade when considering nodes for a web cluster.

We have shown the performance increase realized from adding additional nodes to the cluster. Though the number of back-end nodes we had available was limited, the performance increase per node at maximum throughput is near linear for web service.

There is additional work to be done in the context of CAP policy analysis and development. Our results show that CAP performance is sensitive to the number of Webstone *webclients*, where throughput increases up to a certain number of *webclients*, then flattens out. It would be interesting to see how the system scales with the addition of more back-end nodes, and whether or not RAND and RR still exhibit the same behavior this larger scale.

Still to be determined is the reason behind the degrading performance of CAP relative to RAND and RR as the number of *webclients* increases. Examination of disk and CPU utilization on a single back-end node did not reveal the answer. The system is likely becoming less balanced as the workload increases, and simultaneous data collection of all back-end nodes to compare workloads might be used to verify this possibility. We have shown that nodes built to be identical may still exhibit performance differences that could have an effect on throughput. The CAP algorithm assumes that the pool of back-end nodes which handle a particular category of request is homogeneous. Note that pools of nodes that handle categories exclusive of each other do not have to be homogeneous.

Another possibility for future work is the combination of CAP with real-time server state information. The pure CAP algorithm assumes that the server pool is

homogeneous, as there is no communication between the web switch and the servers as to the current load each is experiencing. Ideally, it would be helpful to allow the use of heterogeneous nodes. As future workload increases, the website operator can add more nodes to the cluster, and the likelihood is that future purchases will result in faster machines. The problem is that slower machines will become more heavily loaded than the faster ones, and the resulting imbalance will degrade the potential performance of the cluster as a whole. Therefore some server state information would be helpful to allow the use of a heterogeneous server pool. The addition of a daemon process that samples one or more of CPU, disk and network utilization on each of the server nodes could relay information on a regular basis to the web switch, allowing for an algorithm that would be a hybrid of CAP and weighted round-robin (WRR). One package available from the open source community that uses server state information is the Linux Virtual Server (LVS) [19], a level-4 distribution package implemented as kernel loadable modules, released under the GNU General Public License.

6 References

[1] C. Yang and M. Luo, “Realizing Fault Resilience in Web-Server Clusters”, Supercomputing 2000: the 13th ACM/IEEE Conference on High Performance Networking and Computing (SC2000), Dallas, TX, 2000.

[2] K. Kant and P. Mohapatra, “Scalable Internet Servers: Issues and Challenges”, Performance Evaluation Review, 5-8, September 2000.

[3] A. Iyengar, J. Challenger, D. Dias and P. Dantzig, “High-Performance Web Site Design Techniques”, IEEE Internet Computing, 17-26, March/April 2000.

[4] V. Cardellini, E. Casalicchio and M. Colajanni, “A Performance Study of Distributed Architectures for the Quality of Web Services”, Proceedings of the Hawaii International Conference on System Sciences, Maui, HI, January 2001.

[5] D. Andresen, T. Yang and O. Ibarra, “Toward a Scalable Distributed WWW Server on Workstation Clusters”, J. Parallel and Distributed Computing 42, 91-100 (1997).

[6] “Guinness World Records 2002”, Time Inc. Home Entertainment, September 2001.

[7] Paul Barford and Mark Crovella, “Generating Representative Web Workloads for Network and Server Performance Evaluation”, Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 151-160, July 1998.

[8] Casalicchio and Colajanni, “A Client-Aware Dispatching Algorithm for Web Clusters Providing Multiple Services”, Proceedings of WWW10, May 2001, Hong Kong.

[9] Casalicchio and Colajanni, "Scalable Web Clusters with Static and Dynamic Contents", Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER2000), Chemnitz, Germany, December 2000.

[10] Mindcraft Software, http://www.mindcraft.com/webstone.

[11] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-Aware Request Distribution in Cluster-based Network Servers", Proceedings of the ACM 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), San Jose, CA, October 1998.

[12] A. Cohen, S. Rangarajan, and H. Slye, “On the Performance of TCP Splicing for URL Aware Redirection”, Proceedings USITS ‘99: The 2nd USENIX Symposium on Internet Technologies & Systems, October 1999.

[13] M. Aron, D. Sanders, P. Druschel, “Scalable Content-Aware Request Distribution in Cluster-based Network Servers”, Proceedings of USENIX 2000, June 2000.

[14] R.S. Engelschall, “Load Balancing Your Web Site”, Web Techniques Magazine, Vol. 3, May 1998.

[15] Apache HTTP Server Project, http://httpd.apache.org

[16] Menasce and Almeida, “Capacity Planning for Web Performance”, Prentice-Hall Publishers, 1998, p. 149.

[17] Pitkow, J., “Summary of WWW Characterizations”, Seventh International World Wide Web Conference, Brisbane, Australia, April 1998.

[18] Teo, Y.M. and Ayani, R., “Comparison of Load Balancing Strategies on Cluster-based Web Servers”, Transactions of the Society for Modeling and Simulation (accepted for publication), 2001.

[19] Linux Virtual Server Project, http://www.linuxvirtualserver.org

[20] Tweedie, Stephen, “Journaling the Linux ext2fs Filesystem”, Proceeding of the 4th Annual Linux Expo (Expo ‘98), Durham, NC, 1998.

[21] Song, Levy-Abegnoli, Iyengar and Dias, “Design Alternatives for Scalable Web Server Accelerators”, Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, 2000.

[22] Cherkasova and Karlsson, “Scalable Web Server Cluster Design with Workload-Aware Request Distribution Strategy WARD”, (not yet published), Hewlett-Packard Laboratories, Palo Alto, CA, 2000.

[23] Standard Performance Evaluation Corporation, SPECweb99, http://www.specbench.org/osg/web99/tunings

[24] Andreolini, Colajanni and Morselli, “Performance Study of Dispatching Algorithms in Multi-tier Web Architectures”, Practical Aspects of Perfomance Analaysis (PAPA) Workshop, ACM SIGMETRICS ‘02, Marina Del Rey, CA, 2002.

Appendix A

Source Code and Apache Modifications

(from psu_webswitch.h)

```
/* ************************************
   psu_webswitch.h

   Globals for shared memory and semaphores
   for use in Apache reverse proxy web switch
   project.

   Robert Jones
   Portland State University
   Portland, OR
   robertj@cs.pdx.edu

   ************************************
*/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <signal.h>
#define NUMSEGS 3  // Number of shared memory segments.  This corresponds
                   //   to how many classes of requests will be received
                   //   by the web server for CAP distribution,
                   //   e.g. static, light dynamic and heavy dynamic would
                   //        be a NUMSEGS value of 3

int psu_numservers[NUMSEGS] = {4, 4, 4}; /* Number of back-end servers for
                                            each category, e.g. {1,3,3}
                                            means 1 node for category 1,
                                                  3 nodes for category 2,
                                                  3 nodes for category 3
                                         */

int psu_numservers_rr = 4; // Should be the total number of back-end nodes used for RR

/* Struct to store integers in shared memory */
struct shmintarr {
    int shmval[NUMSEGS];
};

union semun { int val; };  // Required for Linux SysV semaphores

int psu_sem_id = 0;    // Semaphore handle
int psu_shm_id = 0;    // Shared memory segment handle

key_t psu_ipc_key;     // uniform key to allow multiple processes to find
                       //    the same shared memory segment and semaphore
                       //    array.

char PSU_KEY_CHAR = 'e';  // Additional token used with ftok() call

struct shmintarr *psu_shm_seg;      // Pointer to shared memory segment
                                    //   (a set of integers)

struct sembuf psu_sops[NUMSEGS];

// Attach process to shared memory segment
int psu_shm_attach() {
    psu_ipc_key = ftok("/home/webston/shm.c", PSU_KEY_CHAR);
    psu_shm_id = shmget(psu_ipc_key, sizeof(struct shmintarr), 0666);
    if((int)psu_shm_id != -1) {
        // attach to shared memory segment
        psu_shm_seg = (struct shmintarr*)shmat(psu_shm_id, 0, 0);
        if((int)psu_shm_seg == -1) {
    return -1;
        }
```

Appendix A: Source Code and Apache Modifications - Continued

```
   } else {
        return -1;
    }
    return 0;



}

// Get access to semaphore - this is run once per process instantiation
int psu_sem_get() {
    psu_sem_id = semget(psu_ipc_key, NUMSEGS, 0666);
    psu_sops[0].sem_num = 0;
    psu_sops[0].sem_flg = 0;
    return 0;
}

// Get next value in round-robin sequence
//    This is a critical section subject o
//    race conditions.  Use a semaphore to
//    get exclusive access to this function.
int psu_rr_next() {
    (*psu_shm_seg).shmval[0] = ((*psu_shm_seg).shmval[0] % psu_numservers_rr) + 1;
    return (*psu_shm_seg).shmval[0];
}

// Get next value in CAP sequence.
//   Semaphore should be used to access this
//   function to avoid race condition.
int psu_cap_next(int slot) {
    (*psu_shm_seg).shmval[slot-1] = ((*psu_shm_seg).shmval[slot-1] % psu_numservers[slot-1]) + 1;
    return (*psu_shm_seg).shmval[slot-1];
}
```

```
**************************************************************************************************
(from mod_rewrite.h)
...
#define MAPTYPE_RND 1<<4
/* RMJ - Definitions for RR and CAP policiesi */
#define MAPTYPE_RR 1<<5
#define MAPTYPE_CAP 1<<6
...
static char *select_random_value_part(request_rec *r, char *value);
static void  rewrite_rand_init(void);
static int   rewrite_rand(int l, int h);
/* RMJ - RR and CAP support */
static char *select_rr_value_part(request_rec *r, char *value);
static char *select_cap_value_part(request_rec *r, char *value);
...
**************************************************************************************************
(from mod_rewrite.c)
...
#include "psu_webswitch.h"
...
    // ***********************************
    // **** RMJ - Round Robin Distribution
    // ***********************************
    else if (strncmp(a2, "ror:", 4) == 0) {
        new->type      = MAPTYPE_RR;
        new->datafile  = a2+4;
        new->checkfile = a2+4;
```

Appendix A: Source Code and Apache Modifications - Continued

```
    }
    // ***********************************
    // **** RMJ - Client Aware Policy Distribution
    // ***********************************
    else if (strncmp(a2, "cap:", 4) == 0) {
        new->type      = MAPTYPE_CAP;
        new->datafile  = a2+4;
        new->checkfile = a2+4;
    }
    // ***********************************
...
// ***********************************
// RMJ - Round Robin support
// ***********************************
static char *select_rr_value_part(request_rec *r, char *value)
{
    char *buf;
    int n, i, k;

    /*  count number of distinct values  */
    for (n = 1, i = 0; value[i] != '\0'; i++) {
        if (value[i] == '|') {
            n++;
        }
    }

    /*  when only one value we have no option to choose  */
    if (n == 1) {
        return value;
    }

    /* Index is determined here  */
    if(psu_shm_id == 0) {
        rewritelog(r, 5, "RR attaching to shared memory segment");
        psu_shm_attach();
    }

    if(psu_sem_id == 0) {
        psu_sem_get();
    }

    // get lock
    rewritelog(r, 5, "RR acquiring lock");
    psu_sops[0].sem_op = -1;  // negative == acquire lock
    psu_sops[0].sem_num = 0;  // always zero for RR
    semop(psu_sem_id, psu_sops, 1);

    rewritelog(r, 5, "RR lock acquired");
    // Critical section
    k = psu_rr_next();

    rewritelog(r, 5, "RR releasing lock");
```

Appendix A: Source Code and Apache Modifications - Continued

```
    // Release lock
    psu_sops[0].sem_op = 1;
    semop(psu_sem_id, psu_sops, 1);
    rewritelog(r, 5, "RR lock released");

    /*  and grep it out  */
    for (n = 1, i = 0; value[i] != '\0'; i++) {
        if (n == k) {
            break;
        }
        if (value[i] == '|') {
            n++;
        }
    }
    buf = ap_pstrdup(r->pool, &value[i]);
    for (i = 0; buf[i] != '\0' && buf[i] != '|'; i++)
        ;
    buf[i] = '\0';
    return buf;
}

// ***********************************
// RMJ - Client Aware Policy (CAP) support
// ***********************************
static char *select_cap_value_part(request_rec *r, char *value)
{
    char *buf;
    int n, i, k;
    int category;
    int rc;

    /*  count number of distinct values  */
    for (n = 1, i = 0; value[i] != '\0'; i++) {
        if (value[i] == '|') {
            n++;
        }
    }
    /* The last value is the category of the request
       store as an int
    */
    category = (int)value[i-1] - 48; // e.g. '2' is ASCII 50 decimal
    rewritelog(r,5, "category = %d", category);

    /*  when only one value we have no option to choose  */
    if (n == 1) {
        return value;
    }

    /* Index is determined here  */
    if(psu_shm_id == 0) {
        rc = psu_shm_attach();
        if(rc == -1) {
```

Appendix A: Source Code and Apache Modifications - Continued

```
            rewritelog(r,5, "error attaching to shared memory");
            exit(-1);

        }
    }
    if(psu_sem_id == 0) {
        psu_sem_get();
    }

    // get lock
    psu_sops[0].sem_op = -1;
    psu_sops[0].sem_num = category - 1;  // lock corresponding to our category

    rewritelog(r, 5, "Attempting to acquire lock for category %d", category);
    semop(psu_sem_id, psu_sops, 1);
    rewritelog(r, 5, "Lock acquired");
    // Critical section
    k = psu_cap_next(category);  // get the number of the next server based on
                                 //   the category of request we are serving

    rewritelog(r,5,"next server index based on category %d is %d", category, k);
    psu_sops[0].sem_op = 1;  // positive == release lock
    semop(psu_sem_id, psu_sops, 1);

    /*  and grep it out  */
    for (n = 1, i = 0; value[i] != '\0'; i++) {
        if (n == k) {
            break;
        }
        if (value[i] == '|') {
            n++;
        }
    }
    buf = ap_pstrdup(r->pool, &value[i]);
    for (i = 0; buf[i] != '\0' && buf[i] != '|'; i++)
        ;
    buf[i] = '\0';
    return buf;
}
...
```

Appendix B

Apache Run-Time Configuration Files

```
(httpd.conf.cap)
##
##  httpd.conf.cap -- Apache 1.3 configuration for Reverse Proxy Usage
##  Robert Jones
##  Portland State University
##  robertj@cs.pdx.edu
##  Modified from an example written by Ralf Engleschall
##
User webston
Group PSUPerf

Listen                80
ServerName            192.168.0.134
StartServers          25
MaxClients            256
MaxRequestsPerChild   10000

#   server operation parameters
KeepAlive            on
MaxKeepAliveRequests 100
KeepAliveTimeout     15
Timeout              60
IdentityCheck        off
HostnameLookups      off

#   paths to runtime files
PidFile       /home/webston/rproxy13/bin/httpd.pid
LockFile      /home/webston/rproxy13/bin/httpd.lock
ErrorLog      /home/webston/rproxy13/logs/error_log

#   unused paths
ServerRoot           /home/webston/rproxy13
DocumentRoot         /tmp
AccessConfig         /dev/null
ResourceConfig       /dev/null

#   speed up and secure processing
<Directory />
Options -FollowSymLinks -SymLinksIfOwnerMatch
AllowOverride None
</Directory>

#   enable the URL rewriting engine
RewriteEngine        on
RewriteLog /home/webston/rproxy13/logs/rewrite_log
RewriteLogLevel     0

#   define a rewriting map with value-lists where
#   mod_rewrite randomly chooses a particular value
RewriteMap server cap:/home/webston/rproxy13/conf/servermap.cap
```

Appendix B: Apache Run-Time Configuration Files - Continued

(httpd.conf.cap - continued)

```
#   and make sure no one uses our proxy except ourself
RewriteRule    ^/rproxy-status.*     -   [L]
RewriteRule    ^(http|ftp)://.*      -   [F]

#   Now choose the possible servers for particular URL types
#
RewriteRule    ^/(.*load\.cgi.*)$        to://${server:disk}/$1    [S=5]
RewriteRule    ^/(.*5m.*)$               to://${server:dynamic}/$1 [S=4]
RewriteRule    ^/(.*500k.*)$             to://${server:dynamic}/$1 [S=3]
RewriteRule    ^/(.*\.html)$             to://${server:static}/$1  [S=2]
RewriteRule    ^/(.*\.cgi)$              to://${server:dynamic}/$1 [S=1]
RewriteRule    ^/(.*)$                   to://${server:static}/$1

#  and delegate the generated URL by passing it
#   through the proxy module
RewriteRule    ^to://([^/]+)/(.*)   http://$1/$2 [E=SERVER:$1,P,L]

#   and make really sure all other stuff is forbidden
#   when it should survive the above rules...
RewriteRule    .*                    -             [F]

#   enable the Proxy module without caching
ProxyRequests        on
NoCache              *

#   setup URL reverse mapping for redirect reponses
ProxyPassReverse / http://192.168.0.140
ProxyPassReverse / http://192.168.0.145
ProxyPassReverse / http://192.168.0.146
ProxyPassReverse / http://192.168.0.147
```

****

Appendix B: Apache Run-Time Configuration Files - Continued

```
(servermap.rand)

#
# servermap for RAND policy
#
# Apache mod_rewrite selection table
#
# Robert Jones
# Portland State University
#
# 11/5/01 File created                                Robert Jones
#
# Entry format:
#    entries should be in the form of
#       category_name URL1 | URL2 | ... | URLn
#
#         where category_name is the name of the category, and should
#         correspond to the RewriteRule entries in httpd.conf.
#
#         Integer is 1 for the first category, 2 for the 2nd, etc.
#
#         URL1, URL2, etc. are the URL's for the real servers used
#         for that category.  Note that you may want to assign different
#         categories to different real servers, e.g. a dedicated server
#         might be used only for heavy dynamic requests.
#
#         Note: You must change the entry in httpd.conf to specify
#               the algorithm.  This is just a server map file.  The
#               ServerMap entry must use the appropriate key:
#                   e.g. ServerMap server rnd:......
#                                        cap:......
#               Note that RR policy is simply a 1-category CAP policy.
#
######################################################################
# Unlike the CAP and RR servermap files, RAND does not have an
#    entry at the end of the line specifying a category #

static  192.168.0.140|192.168.0.145|192.168.0.146|192.168.0.147

*************************************************************************************************
(servermap.rr)
...
# One category only for RR - it's called static, but it is used for
#     all categories
static  192.168.0.140|192.168.0.145|192.168.0.146|192.168.0.147|1

*************************************************************************************************
(servermap.cap)
...
static  192.168.0.140|192.168.0.145|192.168.0.146|192.168.0.147|1
dynamic 192.168.0.147|192.168.0.140|192.168.0.145|192.168.0.146|2
disk    192.168.0.146|192.168.0.147|192.168.0.140|192.168.0.145|3
```

Appendix C

Webstone 2.5 Run-Time Configuration Files

(filelist.standard)
@(#)filelist.standard 1.3
Filelist for WebStone 2.5 Standard Run Rules, same as filelist.sample
/file500.html 350 #500
/file5k.html 500 #5125
/file50k.html 140 #51250
/file500k.html 9 #512500
/file5m.html 1 #5248000

****
(filelist.6040.large)
...static files not listed - same ratio as in filelist.standard...
There are 2000 static files, so we need 1333 weight units for a 60/40 mix.
/cgi-bin/cpu_light.cgi 667
/cgi-bin/cpu_med.cgi 400 # This is the 'cpu heavy' service in the thesis
/cgi-bin/disk_load.cgi?file=/home/webston/apache/htdocs/file50k.html 266

****
(filelist.8020.large)
...static files not listed - same ratio as in filelist.standard...
There are 2000 static files, so we need 500 weight units for an 80/20 mix.
/cgi-bin/cpu_light.cgi 250
/cgi-bin/cpu_med.cgi 150 # 'cpu heavy'
/cgi-bin/disk_load.cgi?file=/home/webston/apache/htdocs/file50k.html 100

****

Appendix C: Webstone 2.5 Run-Time Configuration Files - Continued

(from testbed - condensed to options we modified)

```
### BENCHMARK PARAMETERS -- EDIT THESE AS REQUIRED
# Webstone will start running with MINCLIENTS number of processes or threads.
# It will run for TIMEPERRUN minutes.  When that run is finished then the
# number of clients will be incremented by CLIENTINCR and another test will
# be performed.  This will continue until we hit MAXCLIENTS number of clients.
# This entire set of steps will be performed for ITERATIONS number of cycles.
ITERATIONS="3"
MINCLIENTS="2"
MAXCLIENTS="20"
CLIENTINCR="2"
TIMEPERRUN="5"

# This is the host name or IP number of the web server that we will be
# testing.  If you use a host name then be sure your client machines
# can resolve that name.
SERVER="192.168.0.134"

# Port 80 is the default web server port.  If your web server is running
# on another port then you can change this value.
PORTNO=80

# RCP is the command used to copy a file to and from one of the client
# systems or the web server.  For UNIX these can be "rcp" and "rsh" and
# you may have to enable these commands for the machines involved.
# The RCP is used to retrieve configuration files from the web server
# and to distribute test files to the web clients.  If these are left
# empty then WebStone won't attempt to distribute the webclient binary
# and filelist to the clients and you will have to do it by hand before
# running WebStone.
RCP=/usr/bin/rcp
RSH=/usr/bin/rsh
```

Appendix C: Webstone 2.5 Run-Time Configuration Files - Continued

(testbed - continued)

```
# A space-separated list of client machines to use for testing the web
# server.  You can use IP addresses or host names.  If you use host
# names then be sure that the webmaster machine can resolve them.  It
# will try to do an rexec to each of these systems in order to start
# the webclient program.

CLIENTS="192.168.0.135 192.168.0.144"

# These are the user name and password for a user on the client systems.
# The webmaster program will do an rexec to a client system using this
# name and password in order to start the webclient program.
CLIENTACCOUNT=webston
CLIENTPASSWORD=********

# Set this to "true" if we want to use the same random seed during every
# run.  Doing this will make test results more reproducible.
FIXED_RANDOM_SEED=true

# Scratch directory on the client system.
TMPDIR=/tmp

# Full pathname to the webclient program, on the client system.
CLIENTPROGFILE=/tmp/webclient

# Set this to 1 to turn on debugging output.
DEBUG=0
```

Appendix D

Raw Data for Charts

Figure 2

CAP Zero Test
2 clients
4 back-ends

file0k.html 100%

Clients	CAP	RAND	
2	962.11	950.15	values in conn/sec
3	1241.17	1227.12	
4	1366.22	1360.41	
5	1421.91	1422.01	
6	1442.99	1446.36	
7	1444.67	1450.61	
8	1447.72	1449.22	
9	1445.68	1448.53	
10	1444.89	1447.46	

Figure 3

Static Workload
Files copied (2000 files total)
3 iterations, 10 min. run time

RAND	#Procs	Run1	Run2	Run3	Avg	StdDev
	1	237.19	237.52	237.74	237.48	0.28
	5	451.88	452.12	452.31	452.10	0.22
	9	483.54	483.43	483.26	483.41	0.14
	13	496.13	496.18	495.94	496.08	0.13
	17	507.00	513.08	511.70	510.59	3.19
	21	516.18	515.75	514.74	515.56	0.74
CAP	1	231.46	232.19	232.13	231.93	0.41
	5	447.2	447.34	446.65	447.06	0.36
	9	477.41	477.99	477.16	477.52	0.43
	13	491.85	491.72	492.69	492.09	0.53
	17	509.9	508.67	507.3	508.62	1.30
	21	511.04	510.76	511.43	511.08	0.34
RR	1	236.72	237.38	237.36	237.15	0.38
	5	449.41	450.05	449.46	449.64	0.36
	9	481.99	482.38	481.91	482.09	0.25
	13	494.67	495.38	494.97	495.01	0.36
	17	512.07	511.97	512.3	512.11	0.17
	21	515.47	514.35	514.54	514.79	0.60

Appendix D: Raw Data for Charts - Continued

Figure 4

8020 mix

dynamic mix is 80% filelist.standard,
10% cpu_light.cgi, 6% cpu_med.cgi, and 4% disk_load.cgi (50k file as parameter)

RAND	#clients	run1	run2	run3	avg	StdDev
020629_1620	2	103.50	104.19	103.69	103.79	0.36
	4	180.58	178.54	180.27	179.80	1.10
	6	233.07	231.18	233.16	232.47	1.12
	8	266.36	264.15	267.50	266.00	1.70
	10	293.80	289.28	289.92	291.00	2.45
020629_2214	12	307.49	312.77	314.77	311.68	3.76
	14	328.52	320.54	317.59	322.22	5.65
	16	339.54	340.28	336.85	338.89	1.81
	18	353.98	346.28	351.9	350.72	3.98
	20	356.62	366.43	359.43	360.83	5.05
020630_1425	25	381.63	376.93	377.52	378.69	2.56
CAP						
020629_1748	2	105.09	105.81	105.63	105.51	0.37
	4	198.47	198.65	198.57	198.56	0.09
	6	270.00	270.13	269.99	270.04	0.08
	8	315.23	318.01	316.51	316.58	1.39
	10	348.89	347.24	348.92	348.35	0.96
020629_1940	12	369.58	368.92	366.92	368.47	1.39
	14	381.26	377.74	383.72	380.91	3.01
	16	384.53	383.74	382.67	383.65	0.93
	18	388.02	389.80	392.37	390.06	2.19
	20	391.37	395.46	399.15	395.33	3.89
020630_1447	25	377.48	378.14	383.89	379.84	3.53
RR						
020701_0934	2	95.35	104.91	104.28	101.51	5.35
	4	184.76	187.03	186.65	186.15	1.22
	6	240.55	240.46	240.3	240.44	0.13
	8	272.03	273.96	269.24	271.74	2.37
	10	302.1	298.77	298.48	299.78	2.01
	12	302.22	296.05	322.28	306.85	13.71
	14	334.29	335.32	335.16	334.92	0.55
	16	339.06	337.86	343.19	340.04	2.80
	18	348.54	362.99	359.17	356.90	7.49
	20	366.25	360.8	360.25	362.43	3.32
	22	372.52	370.43	379.04	374.00	4.49
	24	380.05	379.45	373.35	377.62	3.71

Appendix D: Raw Data for Charts - Continued

Figure 5

6040 Mix
Files are of same profile as worksheet 8020f (80/20 Mix)

CAP	#clients	run1	run2	run3	Avg	StdDev
020630_2102	2	59.53	59.55	59.68	59.59	0.08
	4	114.94	114.75	115.02	114.90	0.14
	6	161.93	162.45	161.92	162.10	0.30
	8	191.21	191.47	192.17	191.62	0.50
	10	204.43	205.04	206.25	205.24	0.93
	12	203.47	201.34	205.69	203.50	2.18
	14	205.57	208.03	201.96	205.19	3.05
	16	204.79	206.47	209.98	207.08	2.65
	18	208.06	206.32	205.03	206.47	1.52
	20	213.03	207.34	211.32	210.56	2.92
	22	212.31	210.87	209.46	210.88	1.43
	24	209.18	210.01	210.04	209.74	0.49
RAND						
020630_1509	2	58.10	58.03	58.54	58.22	0.28
	4	101.37	100.53	101.26	101.05	0.46
	6	131.79	130.69	131.37	131.28	0.56
	8	147.47	148.28	149.53	148.43	1.04
	10	162.14	161.10	160.28	161.17	0.93
	12	173.05	172.84	169.75	171.88	1.85
	14	182.52	176.95	183.82	181.10	3.65
	16	187.15	184.54	188.50	186.73	2.01
	18	184.23	186.19	190.50	186.97	3.21
	20	194.69	189.92	191.34	191.98	2.45
	22	193.80	196.21	193.84	194.62	1.38
	24	196.79	199.72	201.95	199.49	2.59
RR						
020701_0147	2	59.06	59.08	59.12	59.09	0.03
	4	106.22	107.45	104.31	105.99	1.58
	6	139.08	139.51	138.56	139.05	0.48
	8	160.45	155.87	154.94	157.09	2.95
	10	168.68	168.64	168.43	168.58	0.13
	12	178.85	181.07	180.22	180.05	1.12
	14	185.14	182.38	182.87	183.46	1.47
	16	186.49	188.98	189.08	188.18	1.47
	18	195.39	190.08	188.31	191.26	3.68
	20	197.67	197.91	194.33	196.64	2.00
	22	199.42	197.48	192	196.30	3.85
	24	202.11	201.29	200.79	201.40	0.67

Appendix D: Raw Data for Charts - Continued

Figure 6

Utilization at 10 clients on Wyeast40 - CAP

No.	CPU	DISK
1	64.93	57.47
2	65	66.53
3	67.5	55.2
4	53.9	86.53
5	69.37	46.4
6	66.43	52.33
7	66.4	66.47
8	69.33	53.73
9	66.53	60.4
10	67	57.47
11	67.57	37.93

Utilization at 14 clients on Wyeast40 - CAP

No.	CPU	DISK
1	64.57	44.47
2	66.1	48.4
3	60.13	51.4
4	51.53	47.07
5	50.63	44
6	66.33	62.47
7	65.57	70.87
8	67.87	55.27
9	63.2	55.73
10	66.03	46.93
11	64.87	55.93

Figure 7

Single Server
80% dynamic
20% static
3 min. run time
8 client processes

Node#	Run1	Run2	Run3	Avg	StdDev
34	123	120	124	122.33	2.08
35	124	120	124	122.67	2.31
40	146	152	151	149.67	3.21
44	116	119	117	117.33	1.53
45	146	150	154	150.00	4.00
46	166	166	174	168.67	4.62
47	169	166	163	166.00	3.00

Appendix D: Raw Data for Charts - Continued

Figure 8

6040 Mix on 2 Nodes - Same workload profile as 6040d and 8020e

CAP	#procs	run1	run2	run3	avg	stdev
020701_2353	2	59.03	59.16	59.10	59.10	0.07
	4	95.87	95.21	96.06	95.71	0.45
	6	101.38	102.95	100.42	101.58	1.28
	8	101.22	99.50	98.85	99.86	1.22
	10	101.97	100.70	99.42	100.70	1.28
RAND	2	56.26	56.48	55.54	56.09	0.49
020702_0114	4	80.04	80.77	79.44	80.08	0.67
	6	90.7	89.62	89.72	90.01	0.60
	8	93.71	93.7	93.69	93.70	0.01
	10	94.57	96.99	93.91	95.16	1.62

Figure 9

SMP vs UP

		#clients	run1	run2	run3	avg	stdev
UP x 4	020703_0652	2	56.93	56.71	57.48	57.04	0.40
	020703_0657	4	90.72	91.18	91.52	91.14	0.40
	020703_0702	6	111.42	111.82	111.82	111.69	0.23
	020703_0707	8	124.31	124.00	124.37	124.23	0.20
	020703_0713	10	133.79	132.88	132.96	133.21	0.50
SMP x 2	020703_0859	2	53.58	59.21	59.12	57.30	3.22
	020703_0904	4	93.37	95.83	94.33	94.51	1.24
	020703_0909	6	102.90	100.71	102.03	101.88	1.10
	020703_0914	8	97.91	99.52	102.29	99.91	2.22
	020703_0919	10	100.74	99.26	99.98	99.99	0.74

Figure 10

Test for Scalability from 1 to 4 nodes

CAP (RR)
90/10 Workload

#Proc	4 Nodes	2 Nodes	1 Node
1	17	17	17
3	51	51	35
5	85	69	35
7	118	69	35
9	137	69	35
11	138	69	35