An Evaluation of Caching Strategies for Clustered Web Servers

Thomas Larkin

A dissertation submitted to the University of Dublin,
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science

September 2001

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

_____

Thomas Larkin
14th September 2001

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

_____

Thomas Larkin
14th September 2001

Abstract

The exponential growth of the Internet and the increasing demands put upon Web servers has created the need for a scalable clustered Web servers. In this paper we attempt to analyse the important issues facing the implementor of a clustered Web server. The issues of TCP handoff problems, dynamically generated Web documents and how a Distributed Shared Memory paradigm might be used to optimise a clustered Web server.

We analyse several clustered Web server architectures, and use traces from Web servers in a simulation of several of the Web server architectures to further examine the caching strategies of those architectures. Finally some recommendations are put forth for further research into areas that have been identified as critical to clustered Web servers.

Acknowledgements

I would like to thank my supervisor Christian Jensen for his support and guidance throughout the course of this project. I would also like to thank all my friends, family and classmates for their friendship, support and help during the year.

Contents

Table Of Figures

1 Introduction

In this paper an evaluation of Caching Strategies in the context of a clustered Web server will be examined. Of particular interest is how a clustered Web server using Distributed Shared Memory (DSM) can take advantage of the benefits of the DSM. Cache replacement algorithms will be explored first of all, and then the examination will expand to several clustered Web server architectures.

The architecture of a clustered Web server will be laid out, and the pros and cons of the different approaches will be weighed up. The clustered Web server architectures that will be examined in detail are the Harvard Array of Clustered Computers (HACC), Locality-Aware Request Distribution (LARD), Client-Aware Protocol (CAP), Parallel Pull-Based LRU and Weighted Round-Robin.

A simulation of LARD, PPBL and RR will be reviewed, and the results of this simulation will allow us to gain a better knowledge of what the key areas that need to be focused upon in designing a clustered Web Server.

2 State of the Art overview

2.1 Web Server Caching Strategies

Caching has proved to be an easy and inexpensive way to make the WWW work faster. Web objects such as HTML documents, images, and multimedia clips can be cached in order to improve the response time observed by the Web “surfer”. Caches can be used at three different levels of a typical journey of a Web document from Web server to client: at the Web server itself (in the form of a main-memory cache), somewhere along the server to client path (in a proxy cache) or at the client (a built-in browser cache).

The main-memory cache is of utmost importance to the overall performance of a Web server. It is naturally this area of the Web caching domain that interests us, and it is this that we will now proceed to investigate further. The performance of

the Web server can be increased if requested documents are cached, as this will reduce response time and also decrease server load.

Many existing Web servers (e.g. NCSA and Apache) rely on the underlying file system buffer of the operating system to cache recently accessed documents. This means that a request for a file will involve the operating system buffer copying the contents of the file to the Web server's buffer. This means that we are dependant on the file systems caching, which may not be the best option for Web caching. A better solution is to use a dedicated document cache with a more efficient caching policy (as regards to the particular needs of the web) than that used by the file system.

Data caching in databases and file systems has been the subject of a huge amount of research. However the strategies employed for Web server caching must necessarily be different to those employed in database and file system caching, as the traditional cache policies that perform well in database and file systems do not perform well for the web.

This is due to several key differences in web caching and traditional paging problems. First, web caching is variable sized caching. Web documents vary dramatically in size depending on the information they carry (text, image, video, etc). Web servers always read and cache entire files, whereas file systems and databases deal with fixed blocks of data. Second, caching is not obligatory in Web servers. A file system or database always places requested data blocks in the cache, but a Web server cache manager will not chose to add a document to its cache if this will be detrimental to the performance of the cache. Thirdly, there are no correlated re-reads or sequential scans in Web workloads. A Web user will not re-read the same document in a short space, at least from the perspective of the Web server- all such re-reads will be handled by the Web browser.

These differences demonstrate the differences between traditional caching and the requirements for Web caching. We can see that a single request for a Web document is not a good reason for the cache manager to include it in the cache, as

one access does not mean that this document will be accessed again in the near future. A good cache manager needs to take these factors into account. It should also be noted that Web servers typically deal with fewer data items than file systems and databases do. The number of documents stored in a Web server rarely exceeds 100,000, whereas file systems and databases deal in millions of blocks. This means that the Web cache manager can afford to keep statistics on its documents, to better evaluate which documents should be in the cache in order to improve the overall performance of the Web server.

It is also useful to note the difference between a primary Web server and a proxy server. A proxy server can be potentially drawing from the entire set of Web documents available on the Web, whereas a primary Web server is dealing with much more restricted set of documents. A primary Web server will also have a much more skewed access patterns- there will be a more limited set of popular documents in a primary Web server. It should also be noted that the primary Web server stores its document cache in main memory, and so the caching algorithm employed cannot be too complex, or else the efficiency of the algorithm and the amount of main memory space it consumes will countermand the usefulness of a main memory Web document cache.

The performance metric for Web caches can be either a byte hit rate or a document hit rate. The byte hit rate is the ratio of the number of bytes of data fetched from the cache to the total number of bytes requested. The document hit rate is the fraction of requests satisfied from the cache. In file system and database caches, these two metrics were one and the same, as all data items have the same size. The byte hit rate is important for proxy caches, as it reflects network bandwidth savings. For a primary web server, the document hit rate is of more importance, as it directly affects the response time observed by clients. We will now examine some of the more popular Web caching policies.

2.1.1 Least Recently Used (LRU)

This policy is inherited directly from file system caching. This policy acts upon the premise that recently accessed objects are likely to be accessed again. Thus when a

new request is received, the least recently used document in the cache is removed. While this policy works admirably for fixed-size paging, it has obvious failings when applied to the Web caching domain. This policy ignores the very pertinent issue of documents size.

2.1.2 LRU-Size

This policy [ASAWF95] uses a combination of parameters to select which document to evict from a web cache. Here the largest documents are evicted from the cache first. When documents are of the same size, the least recently used of those documents is removed. This approach does have the drawback of considering one parameter first, and then the second. This means that a document that narrowly "loses" on the first parameter will always be evicted, even if its second parameter is much more in its favour. Using a logarithm of the first sorting key means that the second sorting key is used more often. A variation on this policy which does this is called LRU-($\log_2$ Size) [ASAWF95].

2.1.3 LRU-Min

This policy [ASAWF95] takes the size of the new document into account when evicting documents in the cache. This policy first checks to see if there are any documents of size greater than or equal to the incoming document. If there are, one of these is evicted by LRU order. Otherwise, all documents half the size or greater are considered, and one of more of them is evicted. Then all documents a quarter of the size are considered, until there is enough space for the incoming document. This policy yields better cache performance, but is more CPU intensive than the simpler LRU-Size.

2.1.4 Least Frequently Used (LFU)

This policy uses the frequency of web documents requests to select which document to evict from the Web cache. The document that is accessed least frequently is removed from the cache. Similar to LRU, this policy ignores the issue of document sizes, which can lead to inefficient management of the web cache.

2.1.5 Static Caching

In this policy [TRS97], the cache of web documents is only updated after a certain time. The update is done using the Web server's log to decide which documents should be cached for the next period of time. No new documents can enter the cache during that time period. This policy performs better than most policies given a reasonably predictable flow of requests into the Web server, and has a very low CPU overhead. However its static nature mean that it is always not a suitable policy for Web servers.

2.1.6 Greedy Dual-Size (GDS)

This policy is based upon the LRU model. It has been proven to be online-optimal [CI97]. This means that the GDS policy will perform at least as well as any other online replacement algorithm. The results from [CI97] have shown it to outperform existing replacement algorithms in many performance aspects, including hit ratios. While this policy was introduced as a Web proxy cache replacement policy, it can be easily adapted to Web server caches. As the policy fundamentally uses a cost/size ratio to determine the worth of a document, the maximum document hit ratio can be achieved with this policy by simply setting the cost of every document to 1.

The Greedy Dual Size works as follows. Each cached page has an associated value H. When a page is brought into the cache, H is set to the cost of bringing that page into the cache. When a page is removed from the cache, the value of that page ($_{min}$H) is subtracted from all the remaining values of H in the cache. H is set of the value of cost/size. As this is a main memory Web server cache and not a Web proxy cache, the cost of bringing all documents is set to 1. Therefore H is set to 1/size.

2.2 Caching in a Clustered Web Server

In the context of a clustered Web server, we have to evaluate which of these algorithms will be of most use. Policies which use predetermined fixed values become much harder to implement in a clustered Web server, especially if the Web server is to be scalable. These values have to be determined by running

simulations, and then these values are strongly influential on the performance of the Web cache. When we consider that there will be some sort of distribution policy in a clustered Web server, it can be observed that policies that depend on static values or on complex calculations are not suitable.

Each node in a clustered Web server will have its own Web cache in main memory, and each of these caches will be managed by its own cache replacement algorithm. At this level, the most important factors influencing the choice of cache replacement algorithm are simplicity, performance (document hit ratios) and an ability to adapt its cache in an online fashion. In a clustered Web server each node does not necessarily know the complete request stream being handled by the cluster, it only handles what is sent to it. But in a cluster it is crucial that the nodes are able to adapt to changes in their request streams.

Consider the scenario of a locality based distribution algorithm in a clustered Web server. Requests of type A are sent to node a, type B to node b and type C to node c. Each of the nodes caches will adjust their caches to result in the highest document hit ratio possible. Now consider that node a becomes overloaded, either through an increase in the amount of requests of type A being handled by the server or by node a experiencing a fault of some sort. Now nodes b and c will start to receive requests of type a, as the distribution policy attempts to compensate for the overloaded node a. If their cache replacement algorithms cannot quickly and dynamically adapt to the change in the request stream they have been receiving, the overall performance of the cluster will suffer greatly.

This area of main memory caching replacement algorithms should not be ignored in the context of clustered web servers. The caching replacement algorithm chosen to manage the back end nodes of the cluster will have a significant effect on the document hit ratio. It was noted in [ASP98] that using LRU instead of GDS as a cache replacement algorithm lead to a 30% reduction in throughput results. An interesting experiment would be to analyse the effects of including a combination of online optimal algorithms and more static cache replacement policies, such as Static Caching. However this hybrid approach would require a specialized

distribution policy, and as such is beyond the scope of this paper. It is likely that such an approach would perform better than the approaches tested in this paper, but a problem with such approaches would be a lack of easy scalability.

2.3 Clustered Web Servers

The exponential growth of the Internet has created the need for more and faster Web servers capable of serving over 100 million Internet users. In the past the only solution for scaling server capacity has been to replace the old server with a bigger, faster server. This system has the disadvantage of being very costly in that the investment in the old server is completely wasted once an upgrade is needed. It is in this situation that a clustered Web server is commercially a more viable approach, and imminently more scalable. Rather than replacing the old server, we simply add more nodes to our clustered Web server as demand for the Web service increases.

There are many possible approaches to clustering a Web server. The first requirement that all clustered Web servers must meet is that the cluster must be transparent to client browers- i.e. web browsers are unaware of the server cluster; the cluster must behave in the same manner as a single-server Web server from the point of view of the web browser. Equally, transparency is important for the Web server technology. Early commercial cluster-based Web servers such as Zeus and Inkotmi [F97] did not make the clustering technology transparent to the Web server software. They had an indissoluble whole rather than the layered architecture used in fully transparent clustering. This means that the system needs specialized software throughout, and this increases the cost and complexity of the system. While these technologies outperform traditional single-server Web servers, they are generally dependant on proprietary software.

Most of the research examined in this paper is concerned with clustering techniques that are not only transparent to the Web client, but also to the Web server, as most Web servers do not have any in-built clustering capabilities. We will now examine several high-level views of basic cluster architectures and their pros and cons. The terms used for defining the three separate types of clustering are taken from

[SGR00]. These types are layer four switching with layer two packet forwarding (L4/2), layer four switching with layer three packet forwarding (L4/3) and layer seven (L7) switching with either layer two packet forwarding (L7/2) or layer three packet forwarding (L7/3). These terms refer to which techniques by which the cluster nodes are tied together. In a L4/2 cluster, for example, every node is identical above (OSI) layer two.

Every type of cluster deploys a *dispatcher*, to which all connections to the Web server are initially sent. The servers appear to be a single-server because of this dispatcher. The dispatcher can either be a switch, where it processes incoming data only, or a gateway, where it processing both incoming and outgoing data. Each server will be employing stand-alone Web server software and need not be aware of the states of other server nodes, as Web requests save no state information.

2.3.1 Layer 4/2 Clustering

In L4/2 clustering, the network-layer address is shared by the dispatcher and all of the servers in the pool through the use of primary and secondary IP addresses. All the servers in the cluster pool have the cluster address as a secondary address. This cluster address is the primary address of the dispatcher. All incoming requests for the cluster address are addressed to the dispatcher at layer two.

When the dispatcher receives a TCP/IP connection initiation request, the dispatcher selects a target from the pool of servers to serve the request. This is done by some type of load sharing algorithm, usually a derivative of round-robin (Weighted round-robin, WRR). The dispatcher then updates its tables so that future packets arriving from that connection will be rewritten with the layer two address of the chosen server. The original TCP/IP connection initiation request is then placed back onto the network with the rewritten layer two address.

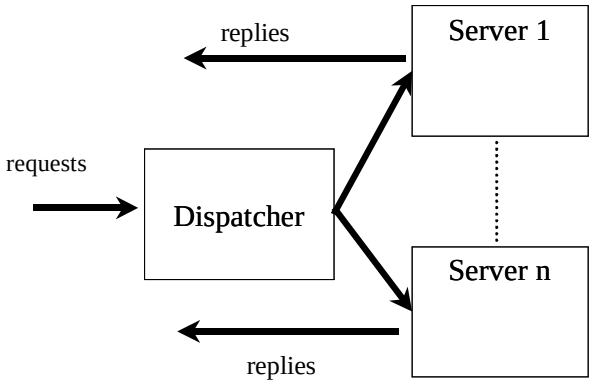

Figure 1- Traffic Flow in a L4/2 Cluster

The advantage of L4/2 clustering is that due to the network address of the server being identical to that that the client originally sent its request to, that server can reply directly to the client. Therefore the dispatcher only processes the incoming data stream, which in Web traffic between client and server, is a small proportion of the total communication between client and server. The dispatcher also does not have to recomputed IP checksums in software because only layer two parameters are modified. The drawback is that all the nodes must have a direct physical connection to the dispatcher, due to layer two frame addressing. This is usually not a problem, as servers in a cluster tend to be on a high -speed LAN.

L4/2 clustering is primarily limited in scalability by network bandwidth and by the dispatchers rate of processing requests, as these are the only part of the transaction that are actually executed on the dispatcher, which is the potential bottleneck in this architecture. There are plenty of examples of this clustering technique in the marketplace, examples being ONE-IP [D97] and eNetwork Dispatcher by IBM [H99].

2.3.2 Layer 4/3 Clustering

L4/3 clustering slightly predates L4/2 methods. In this architecture, every server in the cluster has its own unique IP address. All requests sent to the cluster are addressed to the dispatchers IP address. As above, the dispatcher decides upon a

server to process requests for a new connection through a load balancing algorithm. The dispatcher then rewrites the servers IP address into the packet, and sends it back out onto the network. However the dispatcher will also have to recompute any integrity checks that will be affected, such as CRC and packet checksums.

When the dispatcher receives a packet that is not a connection initiation, it simply inserts the new IP address into the packet, recomputes affected integrity checks and resends out the packet. However packets sent from the servers in the pool to the client must also travel through the dispatcher in this architecture. The dispatcher will rewrite the source address to its own address, and recomputes the integrity codes before forwarding the packet to the client.

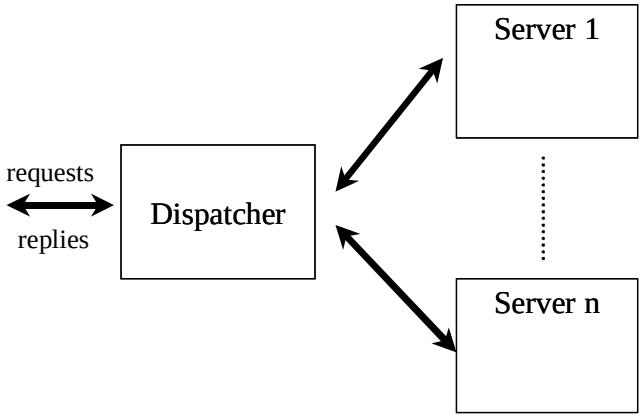

Figure 2- Traffic Flow in a L4/3 Cluster

It can been observed that L4/3 clustering will perform worse than L4/2 clustering, given the added workload imposed upon the dispatcher in L4/3 clustering. Thus the dispatcher becomes much more of a bottleneck, due to the increased demands on its services by the cluster. Examples of L4/3 clustering are Magicrouter [APB96] and LocalDirector from Cisco Systems.

2.3.3 Layer 7 Clustering

The area of L7 clustering is one in which a great deal of research is currently ongoing. These clustering techniques use information contained in the OSI layer seven (application layer) to augment L4/2 or L4/3 dispatching. This is known as

content-based dispatching since it works based on the content of the client request. It is in this area of clustering that both Locality-Aware Request Distribution (LARD) [APS98] and Parallel Pull-Based LRU (PPBL) [C01] fall. Both of these clustering architectures make use of a TCP handoff, an area that must be examined further before examining LARD and PPBL in more detail.

From the introduction to the area of clustering techniques above, we can see that L7 clustering offers the most potential improvements into the performance of a clustered Web server. Content-based dispatching can be used to increase cache hits on the servers, and thus improve the overall performance of the cluster. L7 clustering will still be based upon either L4/3 or L4/2 techniques, but the added complexity of content-based dispatching is compensated by the individual servers in the cluster pool getting more requests that they can serve straight out of their main memory caches.

2.4 TCP Handoff Protocol

When a dispatcher begins to examine incoming requests for content, in order to make locality-aware assignments based on that information, we encounter some problems that WRR avoids by simply assigning requests to the least loaded nodes, irrespective of the content of that request. This is because in order to inspect the contents of a request, a TCP connection must be established with the client prior to assigning that request to a back-end server. This is true for any service (like HTTP) that depends upon a connection-oriented transport protocol like TCP.

In order to circumvent this problem, a technique such as TCP handoff must be used. If a TCP handoff is not employed, then we revert to a situation such as that observed in L4/3 clustering techniques. At best we will have TCP splicing [CRS99], where the data forwarding at the dispatcher node is done in the operating system kernel. While this is an improvement on doing the relaying on a user-level application, it still means that all back-end servers must send their responses to requests through the dispatcher. The TCP handoff mechanism allows the back-end servers to respond directly to the clients without passing through the dispatcher. We will now examine a TCP handoff protocol in more detail.

In [APS98] a TCP Connection Handoff protocol is introduced. The protocol is transparent to the clients and also to the server applications running on the back-end server. Once a connection is handed off to a back-end, incoming traffic on that connection (mostly ACK packets) is forwarded by a forwarding module located at the bottom of the dispatcher's protocol stack.

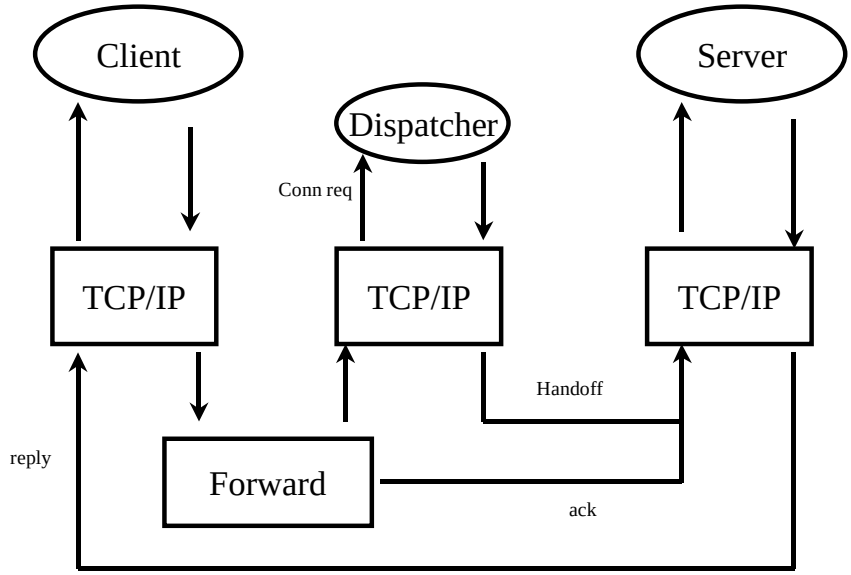

Figure 3 - TCP Connection Handoff

A typical connection scenario is:

1) A client connects to the dispatcher with a request for a web document.
2) The dispatcher at the front-end accepts the connection, examines the request, and then hands off that connection to a suitable back-end server.
3) The back-end server takes over the established connection.
4) The back-end server replies directly to the client.

The dispatcher must still forward all packets coming along that connection to the back-end server, but with this handoff having occurred the dispatcher no longer needs to forward the back-ends responses, as the back-end can now reply directly to the client.

2.4.1 TCP Handoff Protocol Problems

The problem with the TCP Handoff Protocol as outlined above is that for every connection to the cluster, the dispatcher must effectively serialise the state of the

existing TCP connection, and then instantiate a TCP connection with that state on the appropriate back-end server. This amount of work, coupled with forwarding all current connection packets to the back-end serving that connection and making decisions on which back-end to assign new connections to, mean that the dispatcher in this scenario could well quickly become a bottleneck in the clustered Web server as the amount of back-ends in the cluster increases.

Experiments were carried out in [ASDZ00] to investigate the scalability of the TCP handoff protocol described above. Their results show that this architecture does not scale well beyond four cluster nodes. This is obviously not an acceptable limit for a scalable clustered Web server. This problem cannot be easily overcome by simply adding more front-end dispatchers. Adding more dispatchers will introduce load balancing problems on the front-end dispatchers, and secondly most content-aware distribution strategies (like LARD) need centralized control for their distribution policies.

[ASDZ00] goes on to suggest a scalable solution to the problems associated with handing off TCP connections. Examining the dispatcher more thoroughly, we can see that it is in fact doing two services: 1) dispatching the requests to the back-end servers, and 2) interfacing with the client connection and handing off these connections to the back-end nodes. 1) can be clearly termed the dispatcher, as this is the task that implements the request distribution strategy. 2) can be termed the distributor; it is essentially handling the TCP handoffs.

As noted in [ASDZ00], it is this distributor that is restricting the scalability of the cluster, by creating a bottleneck at the front-end node. If this distributor can be assigned to the back-end servers, then the bottleneck of the front-end node is solved. Their experimental results show that the processing overhead for handling a typical TCP handoff is nearly 300µsec, whereas the dispatcher only requires 0.8µsec. Thus distributing the distributor over the back-end nodes will resolve the bottleneck at the front-end of the cluster. Their experimental results show that a centralised dispatcher implementing the LARD policy can service up to 50,000 connections/second on a 300Mhz Pentium II machine. This is over an order of

magnitude greater than the connections/second performance of clusters implementing a TCP splicing or handoff architecture as defined in the previous section.

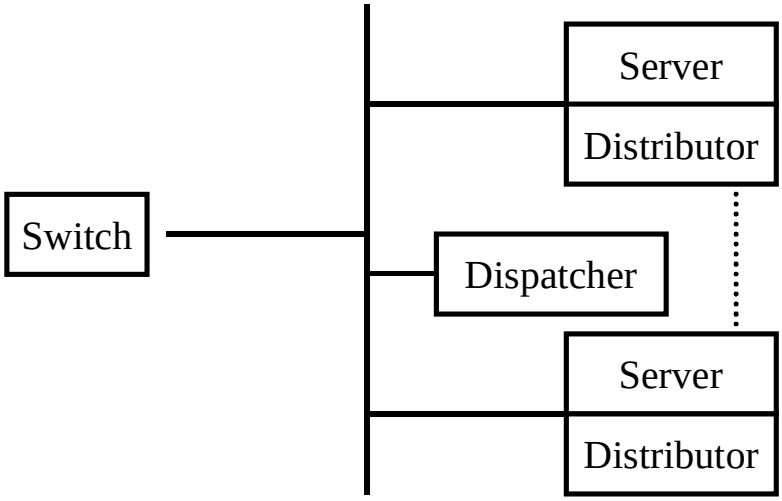

Figure 4- A distributed Distributor Design

As we can see from the architecture suggested in [ASDZ00], the allocation of the various parts of the clustered Web server has changed. We now have a switch accepting requests from clients. This is a layer 4 switch, and can thus be a scalable hardware based switch product. Note that this switch behaves like the front end of a WRR clustered Web server, all it does it distribute requests to back-end servers based on layer 4 information. Therefore at this stage no TCP connections need to be established.

As each connection is a completely separate entity, every connection can be distributed as appropriate for load balancing measures. Once the back-end server receives a connection request, it behaves according to the original TCP handoff protocol, once the destination of the connection has been determined by the centralised dispatcher. This has the added advantage that roughly $1/k^{th}$ (where k is the total amount of nodes in the cluster) of all requests will not need to have a TCP handoff operation, as they will be at the back-end assigned to them.

In this cluster architecture, a typical connection scenario would look like this:

1) The client uses TCP/IP to connect to the chosen distributor (chosen by the switch)

2) The distributor component accepts the TCP connection, and parses the clients request.
3) The distributor contacts the dispatcher to learn which back-end will serve the request.
4) The distributor hands off the TCP connection to the appropriate back-end server.
5) The server takes over the connection, and replies directly to the client.
6) The switch is notified of the handoff, and any subsequent TCP acknowledgements are forwarded directly to the appropriate back-end server.

This proposed architecture solves the bottleneck problem we observed above, when the dispatcher and distributor are both located on the front-end of the cluster. This architecture is a reasonable one for any content-based distribution algorithm, where the TCP handoff problem needs to be addressed. It also has the advantage of being imminently scalable, as each of its components is replicable, apart from the dispatcher component. However further on in this paper we will examine PPBL, which proposes a distributed dispatcher, and the implications of this will have to be evaluated in the context of the proposed architecture proposed above.

2.5 Clustered Web Server Architectures

2.5.1 Weighted Round-Robin (WRR)

This is the commonly used distribution policy used by state-of-the-art commercial clustered Web servers. The front-end dispatcher in this architecture is simply a Layer 4 switch that distributes requests to the back-end nodes based purely on load balancing requirements. The algorithm used is a improved variant of Round-Robin, where the weighting is determined by such factors as current CPU load and disk reading activity, current connections to each node and other factors that vary depending its weighting policies. The front-end dispatcher achieves good load balancing and low idle time rates on the back-end servers. However, as the front-end dispatcher does not open a connection to the client, it cannot make request-based decisions. This means that the TCP handoff problem is avoided, but at a cost.

Before assigning requests to back-end servers, the front-end dispatcher must have a dynamically evaluated weight associated with each back-end server that is proportional to the server load state. This weight is re-evaluated periodically. The three main factors that are taken into account are the loads on CPU, disk and network resources. Typical measures to estimate the load are the number of active processes on the server, mean disk response time and hit latency time, that is to say the average time each request is taking on to complete on that server. These are the factors that a WRR cluster takes into account while calculating load balancing.

WRR clusters have a very high cache miss ratio, as locality is ignored by the front-end dispatcher. This means that the throughput of the cluster is limited by disk accesses, caused by cache misses. The effective size of the cluster cache is that of a single node in the cluster, independent of the number of nodes in the cluster. This is because each cache is completely independent of every other cache in the cluster, and no attempts are made by the cluster to improve the hit rate in these back-end caches. This has the interesting result of the size of each node's cache being a key parameter in how the cluster performs.

The WRR cluster architecture is easily scalable, being bound only by the speed of the switch at the front-end dispatcher of the node. Another interesting problem would be when services other than static Web document requests are required. If the cluster is offering dynamically generated Web documents, or services such as streaming video, the usefulness of the WRR clustered Web server becomes limited. Without a priori knowledge of the type of request issued, intelligent decisions about which node to assign this request to cannot be made. Thus specialized nodes that are dedicated to a certain type of request are of limited use in this environment, as they will be used in a inefficient way (i.e. switch assigns request to back-end node, which then communicates with the specialised service provider, and relays the output of that service provider to its client. As the uses of the Web server become more diversified, and more services are being offered to clients, it can be foreseen that the WRR approach can be less useful. However it must be noted that the WRR clustering approach is a robust, imminently scalable approach, limited

only by the capabilities of the front-end dispatcher, that will always provide evenly balanced response times to all client requests to the cluster.

2.5.2 Harvard Array of Clustered Computers (HACC)

The HACC represents an approach to a clustered Web server [ZBCS99]. We will explore the architecture of this system, and examine its advantages and drawbacks. The HACC cluster is designed to cater for locality enhancement, aiming to improve the relative size of the clustered Web cache (by localising requests to back-end nodes that have already served requests of this nature), load balancing and ease of scalability.

The HACC system employs a “Smart Router” at the front end of the cluster. This Smart Router replaces the Layer 4 switch that would be used in a WRR approach to a clustered Web server. This Smart Router in their implementation can be seen to equate to a front end consisting of a centralised dispatcher and distributor. The Smart Router can be divided into two layers, the High Smart Router (HSR) and the Low Smart Router (LSR). The LSR equates to the low-level kernel resident part of the system, while the HSR represents the high-level application layer decision making part of the system. This is intended to separate mechanism (LSR) from policy (HSR).

The LSR is responsible for TCP/IP connection set up and termination, for forwarding requests to back-end servers and for forwarding documents back to the clients from these back-end servers. The LSR module can be clearly equated with the distributor module defined in the section above. This module is primarily concerned with performance, as it is on the critical path of every request handled by the cluster, and if its performance is not satisfactory, then this module will quickly become a bottleneck for the system.

In their design, the LSR is implemented as a Windows NT device driver that attaches to the top of the TCP transport driver. The LSR listens on the well-known Web server port for connection requests. When a connection request is received, the URL is extracted from the connection request, and passed up to the HSR. The

LSR then enqueues all data from the incoming connection and waits for the HSR to indicate which node to forward this data to. The LSR will continue to ferry data back and forth between the client and the target back-end server until the connection is closed.

The duties of the HSR mean it can be identified as a dispatcher. It is responsible for monitoring the state of the document store and the state of the back-end servers in the system. It uses this information to make decisions about how to distribute requests over the HACC cluster. In [ZBCS99] they have implemented two dispatcher algorithms, one designed to handle requests for static files and one for the document store used by Lotus Domino, which is intended to represent an instance of a Web Application Server (WAS), which generate documents on the fly and require a great deal of compute power on the back-end servers.

The HSR will assign requests to back-end servers in a locality-aware fashion, attempting to improve the overall performance of the cluster by doing so. In the case of static files, this is simply a matter of selecting the server that has been assigned the task of serving requests for that document. In the case of WAS type requests, the issue is more complex. For Lotus Domino requests, assumptions can be made about the request. Lotus Domino requests contain both a Notes Object and a requested action in the URL. The HSR thus attempts to forward requests to back-end servers based on these two parameters.

Load balancing is also carried out by the HSR module. It collects load statistics from each back-end server at periodic intervals. In their prototype the performance metrics utilised are CPU utilisation and bytes transferred to/from disk per second. When the HSR receives requests for data as yet unassigned to a specific back-end, it assigns them to the least loaded back-end. The HSR also attempts to offload a proportion of the documents of an overloaded node onto the least loaded node when such a situation occurs. This is calculated by the load of the overloaded back-end server exceeding that of the least loaded server by a certain amount.

HACC claims to a more scalable cluster architecture than LARD, which we will examine later in this paper. It claims to be able to handle WAS type requests, and so is more suited to a situation where dynamic requests are generated in the system. This is true to a certain extent. There are several disadvantages to this system that we will now explore.

As we explored earlier, in a scalable clustered Web server, the front-end dispatcher/distributor module can easily be overloaded with work, and become a critical bottleneck to the system. This is certainly the case in the HACC design. As the results of [ASDZ00] show, this architecture will quickly reach its limit of throughputs per second, as the Smart Router becomes a bottleneck for the cluster. This was seen to occur with only four back-end servers, and adding more back-end servers does not increase the throughput of the cluster. However, as is suggested in [ASDZ00], the distributor (LSR) could be distributed over the back-end servers, thus freeing up this bottleneck.

The issue of WAS requests and static requests is explored in this architecture. The authors identify that it is a non-trivial task to determine how to combine a set of performance statistics into a single metric that reflects a back-end server's real load in the context of WAS requests. This problem becomes more complicated when a cluster is dealing with a variety of requests for static and dynamic documents. In their design, the HSR appears to handle one type of documents, or the other. The problem is that requests for dynamically generated documents cannot be easily partitioned for locality of reference as static documents can.

Dynamic requests can only sometimes offer the dispatcher information that can be used to make content aware distribution decisions. The problem is that it is a very broad area, and every type of WAS will have different request types, and these will not necessarily be interpretable by a dispatcher in any useful, content-aware way. The HACC system suggests dedicating back-end servers to these type of WAS, which are CPU intensive. For example in one of their scenarios the trace file processed contains a mixture of static file and ASP requests. They suggest

separating ASP requests from static requests, and sending them to different back-end servers.

Their results show that their architecture is most suited to dealing with requests of a dynamic nature, largely because the overhead of the Smart Router is not as much of an issue when the requests incur a CPU compute on the back-end servers. Their architecture does not appear to be satisfactorily scalable, especially when a mixture of dynamic and static requests are considered. The solution they suggest for this scenario is at best an ad-hoc solution, and while it would work, would require much experimentation on the cluster to achieve the appropriate balance of back-end servers that can handle either static or dynamic requests.

The HACC cluster architecture thus has introduced some interesting problems into the area of clustered Web servers. We can see that the issue of dynamic requests, and how to handle them is a non-trivial one. We have seen that dynamic requests bring an interesting problem to content-aware distribution policies. If the WAS requiring dynamic requests can provide information that the distribution policy can interpret from the URL of such requests, then it can make some attempt to distribute the requests in a locality-aware way. However this information is not always available. Possible solutions to the problem appear to be either to have dedicated back-end servers to deal with these type of requests or to allow all back-end servers to handle these requests.

2.5.3 Client-Aware Dispatching Algorithm (CAP)

The Client-Aware Policy (CAP) is a dispatching policy motivated by a desire to improve load sharing in Web clusters that provide multiple services such as static, dynamic and secure information [CC01]. This is an interesting approach to dealing with the problem of diverse services being offered by a clustered Web server, and is worth looking into in some detail.

The authors of [CC01] argue that as the heterogeneity and complexity of services and applications provide by Web sites continuously increases, so too must the clustered Web server adapt to deal with these increasing demands. They argue that

content-aware dispatching policies that wish to handle requests in a heterogeneous and dynamic system require expensive mechanisms for monitoring and evaluating the load on each back-end server, as they gather results, combine them into meaningful measurements and make real-time decisions based on these measurements. They propose a dispatching policy based only on client requests.

The client-aware policy depends upon classifying incoming requests. In [CC01] they classify Web services into four categories.
Web publishing: Sites that provide static information and dynamic services that do not intensively use server resources. The content of dynamic requests is not known at the instant of a request, but the results are generated from database queries whose arguments are known beforehand.
Web transaction: Sites that provide dynamic content generated from possibly complex database queries. This is a disk bound service as it makes extensive use of disk resources.
Web commerce: Sits that provide static, dynamic and secure information. Security may be necessary, and so use of the SSL protocol is expected. Web commerce services are disk and/or CPU intensive.
Web multimedia: Sites providing streaming audio and video services. These services are not considered in the paper as these services are mostly offered by specialised servers and network connections.

The basic idea behind CAP is that although the dispatcher cannot estimate the precise amount of time a request will take, especially when requests are of a disk-bound or CPU intensive nature, it can distinguish the class of the request, and its impact on Web server resources. CAP classifies the above classification into four classes: static and lightly dynamic Web services (N), disk bound services (DB), CPU bound (CB) and disk and CPU bound (DCB). The dispatcher recognises incoming requests as belonging to one of the classes above (as this policy is a content-aware policy), and maintains a circular list of assignments for each class. The dispatcher thus works in a Round Robin fashion, attempting to keep the loads of the back-end servers balanced by assigning requests of the various classes to the back-end nodes in a circular fashion.

This is an interesting approach to the problem of dynamic and varied Web services being provided by a clustered Web server. CAP attempts to balance the load over the back-end nodes in a more intelligent, proactive (rather than the reactive approach taken in WRR for example) manner. CAP has the advantage of not requiring hard tuning of parameters required by most dynamic policies. As we argued above, this is a valid difficulty facing implementations of clustered Web servers. A cluster where a central dispatcher makes load balancing decisions based on parameters collected from the back-end servers at regular intervals, and attempts to exploit locality in its cluster by partitioning the work set, will encounter real problems when Web services that demand high CPU and disk activity are introduced. The problem is such policies will require a high amount of tuning of parameters to be able to work correctly. The CAP system does not require this, once the requests have been classified correctly.

A drawback of this system is that requests that normally could be dealt with quickly will not, in a system employing CAP. This is because CAP is strongly concerned with load balancing, and while it is a content-aware policy, it chooses to not use this information to its full potential. Clustered Web servers employing CAP will have to deal with the TCP handoff problem, and the additional overhead that adds to the system. CAP can be more equated with WRR than other content-aware distribution strategies in fact. While CAP incurs the penalties involved in a Layer 7 dispatcher, it only concerns itself with load balancing. While WRR does this in a reactive fashion, CAP uses classifications of requests to attempt to balance the loads of the back-end nodes in a proactive manner. While the CAP introduces some interesting issues into the area of dispatching polices, it is doubtful whether this is the “silver bullet” to a clustered Web server providing multiple Web services.

2.5.4 Locality-Aware Request Distribution (LARD)

The locality-aware request distribution (LARD) strategy [APS98] examined below is a form of content-based request distribution, focusing on improved cache hits amongst the back-end servers in the clustered Web server. The motivation behind this strategy was a determination to create a content-aware distribution strategy,

that took into account the service/content requested and the current load on the back-end servers when deciding which back-end to allocate a given request to. This strategy is similar to the HACC dispatching strategy discussed above, however the algorithms employed for attempting load balancing and dispatching techniques are not as dependent upon commercial products for measuring load balancing. The challenges identified by the authors were creating a strategy that simultaneously achieve load-balancing and high cache hit rates on the back-ends, and the creation of a protocol that allows the front-end dispatcher to hand off established client connections to a back-end server. The secondary of these challenges (TCP handoff) has been explored above, and the distributed architecture suggested in [ASDZ00] was designed with LARD in mind.

The authors of LARD acknowledge the need for clustered Web servers to be capable of representing a cache that is greater than that on each of the back-end's main memory cache. This is because as the demands of what a Web server can do increase, so to do the demands on the caching of that Web server. Round-robin distribution effectively limits the cluster's working set to that that can fit into a single main memory cache. With LARD the effective cache size approaches the sum of the back-end's cache sizes. Thus adding extra back-end servers to a cluster of this nature can effectively increase the working set of that cluster as well as accommodating increased traffic (by having additional CPU power).

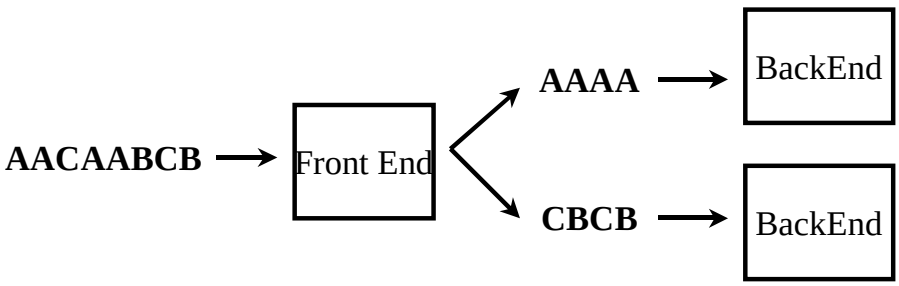

Figure 5- Locality-Aware Request Distribution

The basic LARD algorithm works as follows. The dispatcher maintains a one-to-one mapping of targets to back-end servers. When a request comes in for a new target, it is assigned to the least loaded node in the cluster. Load in LARD is

measured by the amount of current connections to a back-end server that have not yet been completed. This is quite a simple method of measuring load in the back-end servers, but has the advantage of not requiring many CPU calculations to collect and compute. Servers with idle CPU time will tend to have active connections approaching zero, while overloaded servers will have an increasing amount of active connections. By monitoring the number of active connections to a back-end server the dispatcher can estimate the relative load on a back-end without having to communicate explicitly with the back-end servers. This is in contrast with WRR and the HACC system, where the front-end dispatcher regularly polls the back-end nodes for data in order to compute the relative loads on those back-ends.

LARD partitions the working set as requests are entered into the cluster, dynamically distributing the this working set over the back-end nodes. LARD aims to only reassign targets when there are back-ends going idle while other back-ends become overloaded. Re-assignment will naturally result in cache misses on the new node, and so re-assignment should not be done to balance out temporary load imbalances. T_{low} is defined as the load (number of connections) below which a back-end is likely to have idle resources. T_{high} is defined as the load above which a back-end is likely to cause substantial delays in serving requests. Once a back-end reaches $2T_{high}$ a target will automatically be allocated to a lesser loaded back-end, even if no back-end has a load of less than T_{low}.

The front-end also limits the amount of connections allowed concurrently in the cluster, so as to prevent the cluster from behaving like WRR as the load on all nodes rises towards $2T_{high}$. To avoid this, LARD limits the total sum of connections to the value of S, where $S= (n-1) * T_{high} + T_{low} –1$. *n* being the number of back-end server nodes in the cluster. Setting S to this value means that at most *n*-2 nodes can have a load >= T_{high}, while no node has a value less than T_{low}. S also ensures that enough connections are admitted to ensure that all *n* back-ends can have a load greater than T_{low}, and still have room for limited imbalance between the back-ends (and therefore avoiding continual re-assignment of targets on “overloaded” back-ends).

Basic LARD can be improved by using replication. Replication is desirable to allow the cluster to assign more than one back-end to be served for popular requests. LARD with replication differs from basic LARD as follows. The front-end dispatcher maintains a mapping of a target to a set of back-end servers. Requests are passed to the least loaded back-end in that set. When a load imbalance occurs the dispatcher checks to see if the requested documents server set has changed recently (within *t* seconds). If it has, the dispatcher selects a lightly loaded back-end and adds it to the server set of that target. If the server set has not changed recently (within *t* seconds) and has multiple servers, then the dispatcher removes a back-end server from the server set of that target. This ensures that a target that was popular in the past but is not anymore is not served by several back-ends.

[APS98] provides results that show considerable performance improvement on state-of-the-art WRR. The achieved throughput with a working set that does not fit into a single back-end's main memory cache with LARD exceeds that of WRR by a factor of two to four. It must be noted however that all these results are delivered from tests with static documents. The performance of LARD with dynamic requests was not tested in their test bed. In tests against CAP [CC01] using traces containing 20% dynamic documents, the LARD policy falls 16 to 21% short of the performance of CAP. This is not surprising, considering that the LARD policy as it stands makes no real attempt to deal with dynamic documents, which will increase the load of a back-end server dramatically, without this becoming apparent to the load balancing attempts of the front-end dispatcher. This is because the loads are measured by the amount of active connections to each back-end, and does not take into account that some of these requests might be of a dynamic nature.

As is suggested in [ZBCS99], LARD might be able to handle dynamic requests in a more satisfactory way if weighting was given to the dynamic requests, so that a back-end node serving dynamic requests will show this in its weighted load. This methodology would however mean that extensive testing and fine tuning would have to be done to determine how much weighting should be given to each type of dynamic requests, as is argued in [APS98]. Perhaps a LARD-driven cluster serving

both dynamic and static requests would need to have its back-end nodes partitioned into sets that serve static requests and sets that serve dynamic requests. However more research is needed into the nature of dynamic requests, and to what extent dynamic requests can also be cacheable before the issue of clustered Web servers serving offering traditional static Web documents and the more diverse array of services on offer from Web Application Services (WAS) can be addressed more fully.

2.5.5 Parallel Pull-Based LRU (PPBL)

The PPBL request distribution algorithm suggested in [C01] is an interesting variation on the clustered Web server architectures suggested above. Whereas all the distribution algorithms explored so far are push-based algorithms, PPBL (as its name suggests) is a pull-based distribution algorithm. The logic behind this decision is an attempt to distribute the dispatcher over all nodes, and by doing so to relieve the front-end of this task. The reason that this approach is justified in [C01] is that the architecture of their cluster is based upon a Distributed Shared Memory (DSM) system using a Scalable Coherent Interface (SCI) memory mapped network. This DSM should be utilised by PPBL in order to improve the decision making and general overall performance of the distributed dispatcher.

The PPBL dispatching algorithm introduces several new features not yet seen in the Web server clusters investigated above. While the architecture proposed in [C01] is designed for a clustered Web proxy cache, its architecture is equally valid for a clustered Web Server. This is mainly due to the fact that the approach suggested is a simplistic one, and thus did not need to be optimised for a Web proxy cache environment. While the designers of the system envisaged the system compressing and uncompressing files before sending them to the clients, this scenario is not implemented in their prototype, and therefore their results are equally valid for a clustered Web server as they are for a Web proxy cache. As we have outlined above, the performance of the cluster's total cache is of vital significance of the performance of the clustered Web server as a whole, and so the architecture they have put forward is just as valid for use in a clustered Web server.

The architecture of the system is quite a simple one. The task of the front-end has been once again reduced. Now that the job of dispatching is going to be distributed over the back-end servers, the task of the front-end is much reduced. If we employ the scalable architecture suggested in [ASDZ00], then the front-end is also not responsible for the distributor tasks. Therefore in this architecture, it is anticipated that the front-end will not become a bottle-neck for the system.

The system consists of a front-end and back-end nodes as usual. The front-end manages an incoming queue of requests, and the back-end nodes each have a section of the working set of documents that they own in a URL table and a LRU cache of these documents. The URL table consists of the names of the documents that the back-end owns, and a three-slot LRU providing the identifier of the last nodes that served this request (called its serving LRU).

Incoming requests to the cluster go through a front-end as normal. The task of the front-end is to insert the requests into an incoming queue. The back-ends search this incoming queue for requests that are not yet found. The back-end performs a search of its URL list to check if it owns it or not. This search is done in LRU order. If a back-end node does own the request, then it updates the incoming request queue entry with its node identifier. At this point the back-end node has signalled to the front-end that it is handling this request, so the request is then added to the work queue of that back-end node.

If a request is not found to be in the current contents of any back-end nodes, then the front-end chooses the least loaded node, and assigns the request to that back-end node. That node then updates its URL table accordingly. A Status table is also maintained, containing the current loads of all the back-end nodes. This is regularly updated by the back-end nodes. In the PPBL system, load is estimated by the amount of data it still has to serve. The least loaded node is the one with the lowest amount of data to serve.

If a back-end has found that it owns a request, but is overloaded (according to some static threshold), then the node to which this request will be passed is determined

from the URL table entries' serving LRU. If this is empty, then the least loaded node is chosen and added to the serving LRU. If the back-end is the least loaded node in the cluster, then it keeps the request. Each node's main memory cache is run by the LRU cache replacement algorithm. This is an aspect of this architecture that could be easily enhanced by using Greedy Dual Size or another of the more optimised versions of LRU, and this is acknowledged by the authors of this paper.

The implementation of this architecture involves a great deal of synchronisation between the dispatcher modules on the back-end nodes and the front-end. This is done through a producer/consumer model using the distributed locks provided by the DSM and its components. The incoming queue simply resides in the DSM, and is thus accessed quickly and easily by all nodes in the cluster. Each client connection is handled by a dedicated thread in the front-end.

Every incoming queue entry has one start and one end lock. The request is first put in the queue. Then the start lock is released to make the backends start the lookup. Then the dedicated thread waits for the end lock for lookup completion. This is the rendezvous with the back-end nodes. If none of the back-ends claim the request, then it is added to the least loaded node as mentioned above. Once the request has either been claimed or assigned, then the request can be removed from the incoming queue.

The backend process can also be multithreaded to allow multiple lookups. The lookup thread first waits for a request to become available (the consumer waits for the producer). Then the thread performs the lookup, and if it is found, the thread inserts its node identifier in the incoming queue request, to acknowledge this. The end of the lookup is signified by releasing the end lock.

This is the basic implementation of the architecture. The authors suggest further improvements to optimise the performance of the algorithm. A completion queue and index table are added to the front-end. The completion queue has the same number of entries as the incoming queue. When a back-end has checked with a request *i* in the incoming queue, it performs an atomic fetch and increment on

completion queue *i*. If this value is equal to the number of back-end nodes minus one, then this node was the last node to lookup that request, and then it is only the duty of this node to release the end lock. This process will reduce the number of locks required to be released per incoming queue request.

The index table is used to solve the lookup starting problem in the saturated case. If there are more requests to check in the incoming queue, then there is no need for each thread to wait for the start lock to be released. Therefore each node indicates which is the next request it is going to treat, including the front end. If the index of the back-end node is greater than the front-ends index, this means that it must block and wait for a new request to be inserted into the i ncoming queue.

The PPBL system can be defined as a content-aware approach to clustered Web servers. In contrast to the centralised dispatchers we have encountered in every other system so far examined, we here have a distributed dispatcher. The TCP handoff problem is not explored in detail in this paper, apart from commenting that the SCI architecture could be used to improve the performance of this protocol. The TCP handoff remains an issue that PPBL has to deal with, as the contents of requests are examined before a decision is made as to which server node will deal with that request. Therefore we can assume that an approach similar to that suggested in [ASDZ00] will be taken, with distributed distributors as well as dispatchers being implemented in the system.

The issue of load balancing has also been examined by this approach. It can be seen to be more of a reactive approach to load balancing, rather than the proactive attempts made by LARD, for example. However, it is interesting to note that the load balancing evaluation is carried out on the back-end node in question, which is something that does not occur in any of the other systems. This has the advantage of while still being a reactive system, it is imagined that overloading will be detected quicker in this system. This is because while other systems use a timer to collect load information from the back-end nodes, here it is the back-end node itself that determines when it is overloaded.

[C01] produces some early results of the PPBL system. They report a steady increase in the throughput per second as the number of nodes in the system increases. However this paper does not measure the document hit rate achieved in this system, a measurement that is vital in an efficient Web server.

2.6 Distributed Shared Memory Based Clustered Web Servers

The past decade has seen a two-order-of-magnitude increase in processor speed, and only a two-fold increase in disk access time. The gap between disk and processor speeds has been steadily increasing, and will continue to do so. There have been significant improvements in the network speeds of Networks of Workstations (NOWs). This means that the network is now considerably faster than disk for transferring data to and from main memory, a performance enhancement due primarily to the fact that remote-memory accesses avoid the expensive seek and rotational latencies associated with disk access [F96].

In a Web server, the disk access times are reduced by the use of an efficient cache. In a clustered Web server, the Web cache becomes a distributed and perhaps partitioned object, depending on the design of the clustered Web server. The amount of requests for Web documents that involve a disk access could be reduced by having a more globally-aware cache management system. In most current clustered Web servers once a request reaches a node, that node relies solely on its own cache to provide it with as efficient as possible a performance that reduces the disk access times to the absolute minimum. However, a cluster utilising a global shared memory model could well improve on the performance of its nodes by taking advantage of the benefits that the shared memory model provides by fetching a document stored in a remote node's cache rather than fetching it from disk.

The shared memory model that we will be examining is Kaffemik [AWCJC00]. Kaffemik is a distributed Java virtual machine (JVM) that runs on a cluster of workstations interconnected by a Scalable Coherent Interface (SCI) [IEEE93].

Kaffemik is based upon a small area network using a Single Address Space (SAS) system to support a shared object space. This provides us with a clustered architecture that will support remote accesses to other nodes. [CKP98] demonstrates that a page could be accessed over 50 times faster from remote memory than from a local disk. As Kaffemik is implemented on SCIOS/SCIFS [KHCP99], this is a significant and interesting statistic.

3 Simulation design and requirements

We have examined some clustered Web server architectures. Most of them are not designed for a cluster employing a shared memory model. PPBL is the exception here, as it is designed specifically for such an architecture, and indeed takes advantage of specific services provided by the API of the system it is implemented on to improve performance. These improvements are mostly involved with synchronisation, and not with utilising remote nodes to provide the cluster with a more intelligent, globally-aware caching algorithm.

A simulation was required to examine the behaviour of clustered Web server architectures, and to examine the behaviour more specifically of the caches in the cluster. The architectures that we decided to simulate were PPBL, LARD and a simple RR approach. PPBL was selected because it is an architecture that is designed for a shared memory environment, LARD because it outperforms most other clustered Web server architectures [APS98], and a RR system to act as a benchmark for the other two architectures. We are not interested in achieving a realistic throughput of requests per second, but rather on examining whether the Distributed Shared Memory (DSM) architecture is a useful one for implementing a clustered Web server.

In all of the architectures we have examined, there has been no attempt at managing the “global” cache in any way. That is to say that when a node decides to process a request for a document, the fetching of that document is done either from the node’s cache or from a local disk. Each cache is local and ignorant of the contents of the other caches in the cluster. This ignorance is justified by LARD and

PPBL by taking advantage of content of requests before allocating them to nodes. Therefore a node's cache is solely interested in a sub-set of the total working set of that Web server. Seen in this way, it does not seem necessary to increase the complexity of the cache management algorithm by attempting to have a more cooperative approach to the management of the caches on the cluster.

The crux of the problem is whether a DSM based cluster architecture could benefit from a more globally-aware caching strategy, given the great optimisations that can be gained from using remote memory access instead of disk access. To decide upon this, a simulation to examine the performance of the node's caches and amount of disk access in a clustered Web server will be implemented. This simulation will provide us with statistics from which we can determine whether a globally-aware caching strategy would be of benefit to a clustered Web server using a DSM.

Another possible benefit of a DSM-based clustered Web server is that not all nodes need to have the entire working set of the Web server in their local disk space. As long as one node on the cluster has a copy of a document on disk, then any node can request and read this document into their own cache without too much delay. As the working set of a Web server can be quite large, this could provide a useful service to the clustered Web server as a whole.

4 Simulation implementation

The simulation has been implemented in the Java programming language. All the constructs used in creating simulations of LARD, PPBL and RR have been either been built from scratch or utilising existing constructs from the Java API. The primary objective of this simulation is to observe the behaviour of the caches in the three architectures, and whether any of these designs would be suitable for implementing a clustered Web serving using a DSM architecture.

This simulation model is attempting to investigate cache performances on the nodes in the cluster. Therefore the overall throughput per seconds achieved in the simulation is not of interest to us. This statistic would be of interest in a prototype

environment, where Web server applications would be running on the nodes of the cluster. In this simulation, the nodes only check to see if the document requested is in the cache. We are also not concerned with the TCP Handoff problem in our simulation, as there are no real requests (and therefore no attempted connections) to our simulation. The simulation uses a Web trace to provide a realistic list of requests that would be put to a Web server, and the performance of the caches in the clustered Web server is then observed.

The basic process that runs is as follows. A trace (as discussed below in Section 5) is processed and transformed into a file of `Request` objects. This file is input as a stream of objects that is received by the `FrontEnd` of the cluster. The `FrontEnd` implements a distribution algorithm as a thread, called `DistribAlgo`. The `DistribAlgo` decides which `Node` receives the `Request`. The Request is passed to the `NodeManager` in charge of that request, who will in turn pass it on to the Node. Each `Node` in turn implements a `CachingAlgo`, which also runs as a thread on that `Node`. The `CachingAlgo` is the cache replacement algorithm. Every `Node` also contains a `Cache`, which contains a list of the current documents stored, and also a `StatsObject` for each document that has been requested at that node. The `StatsObject` contains information such as how many times that document has been requested, and how many times it was in the cache when requested (e.g. a cache hit).

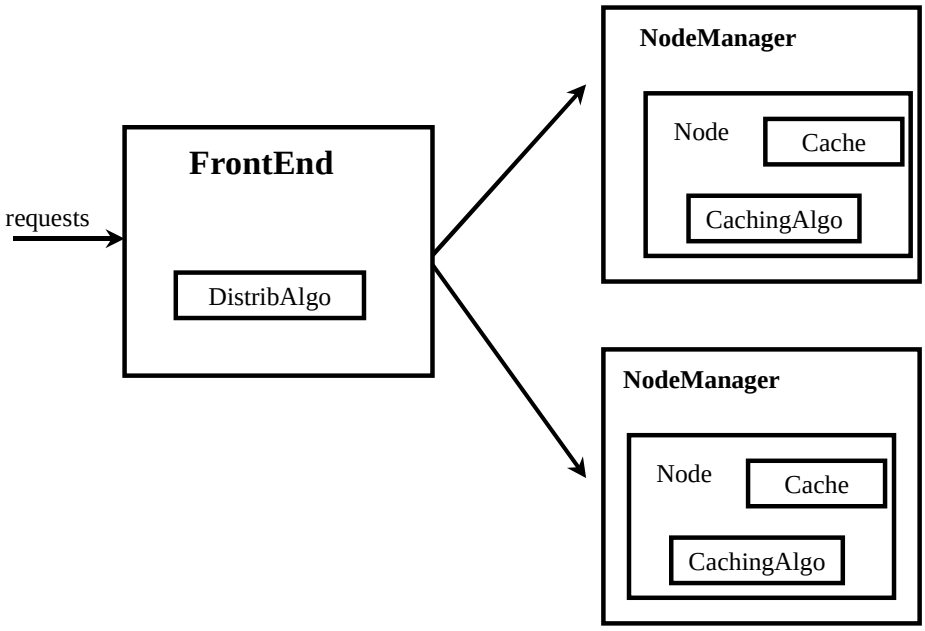

Figure 6- Basic Simulation Design

The basic architecture of the cluster is represented by a `FrontEnd` through which all `Requests` are processed. The number of `Nodes` in the cluster is configurable as is the size of the `Cache` on each of the `Nodes`. In our simulation, the amount of nodes contained in the cluster varies from two up to sixteen. The cache sizes have been kept constant throughout the simulations at 1M. As we will see this produces a small aggregate cache size for the whole cluster as compared to the total size of the working set (see the Section 5 for more details), especially when the cluster contains a low amount of nodes. The small cache sizes will mean that the caches will be put under a great deal of demand, as the small cache attempts to handle the large amount of diverse requests coming into it.

4.1 RR Simulation

The first simulation that was implemented was the RR distribution architecture, using the GDS cache replacement algorithm on each node. This architecture was relatively simple to implement. The `FrontEnd` implemented the `RRDistribAlgo`. This distribution algorithm simply assigned `Request` objects as they arrived in the `FrontEnd` in a round robin fashion. The `NodeManager`

passes the `Request` to the `Node`, where the `GDSAlgo` decides which document needs to be replaced from the cache by popping the first `GDSobject` (or if the document is already in the cache, then its GDSobject is removed) from the `PriorityQueue`, and then that `GDSobject` is replaced into the `PriorityQueue` at the appropriate point depending on the value of the object, as calculated according to the GDS formula explored in Section 2.1.6.

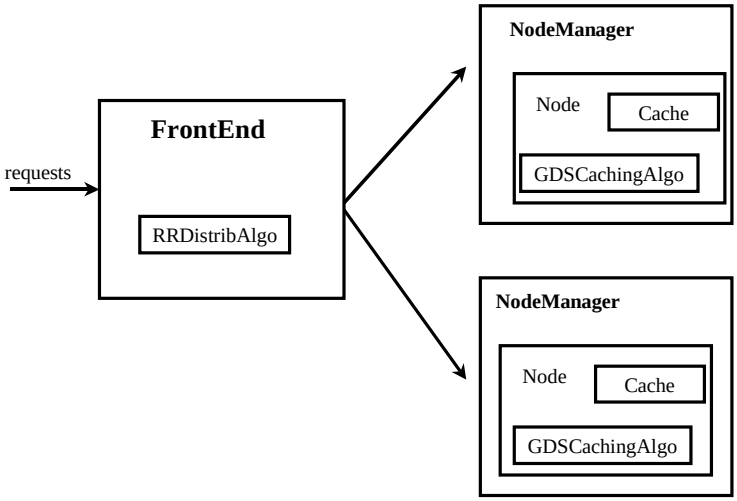

Figure 7- Round Robin Architecture

4.2 LARD Simulation

The LARD simulation demanded a more complex implementation than RR. This is because of the load balancing measurements needed to run LARD successfully. As before, the `FrontEnd` accepts `Request` objects. However as each `Request` is a separate connection, the number of connections is limited to *S*, as mentioned above (see Section 2.5.4). The LARD algorithm checks to see which `Node` has been assigned this `Request`. If this `Request` has not been requested before, then it is assigned to the least loaded node. This is determined from the `LeastLoadedQueue`. The `LeastLoadedQueue` is a sorted list containing the load balancing information about all the nodes in the cluster (current connections to each node- in other words, all the `Request` objects that have yet to be processed on that particular `Node`). Once the `Request` has been assigned a `Node` it is passed onto that node's `NodeManager`. Each `NodeManager` contains a `ConnectionQueue`, an `AddThread` and a `RemoveThread`. The

`ConnectionQueue` is the list of requests that this node has yet to process. When the `AddThread` inserts a new `Request` into the `ConnectionQueue`, the `LeastLoadedQueue` is updated to reflect this.

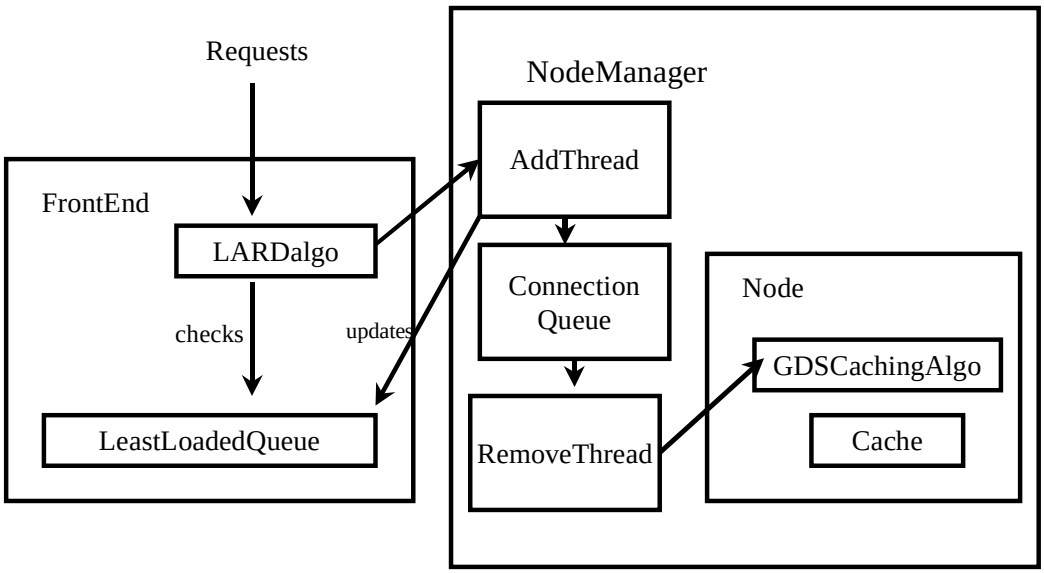

Figure 8- LARD Architecture

The `RemoveThread` takes out the first `Request` object from the `ConnectionQueue` as they are processed by the node's cache replacement algorithm (this is GDS in the case of LARD). As each `Request` is processed by the `Cache`, the relevant statistical information is inserted into that document's `StatsObject`.

The difficulty in implementing this system is configuring the correct values for T_{high} and T_{low}. If these figures are too low, then LARD becomes WRR, and if they are too high, then the system allows nodes to be overloaded while other nodes in the cluster remain idle. These figures have to be adjusted to the LARD implementation by trial and error, and depend upon the performance of the system they are a part of. It is also necessary to keep a count of the total current connections in the system. This information needs to be accurate and up-to-date, as the system ceases to accept connections when the upper limit of *S* is reached. In our simulation, this information was retrieved by polling the `LeastLoadedQueue`. In a more realistic implementation of the LARD architecture, each node will know

how many connections it has open, and this information should be easily accessible by the LARD algorithm.

4.3 PPBL Simulation

This simulation was the most difficult to implement. There is necessarily a lot of synchronisation involved in this architecture due to the distributed Dispatcher module. While this synchronisation utilises the DSM's synchronisation protocols in [C01], in this simulation none of these methods are available. As synchronisation in Java is done through the basic technique of acquiring and releasing of monitors, this added a lot of complexity to the simulation model. The PPBL simulation finally produced was not an optimal implementation. However, the model did avoid deadlock, and so while slow, it still produced valid results, in that the distributed Dispatcher did dispatch requests to the appropriate nodes, as set forth in [C01].

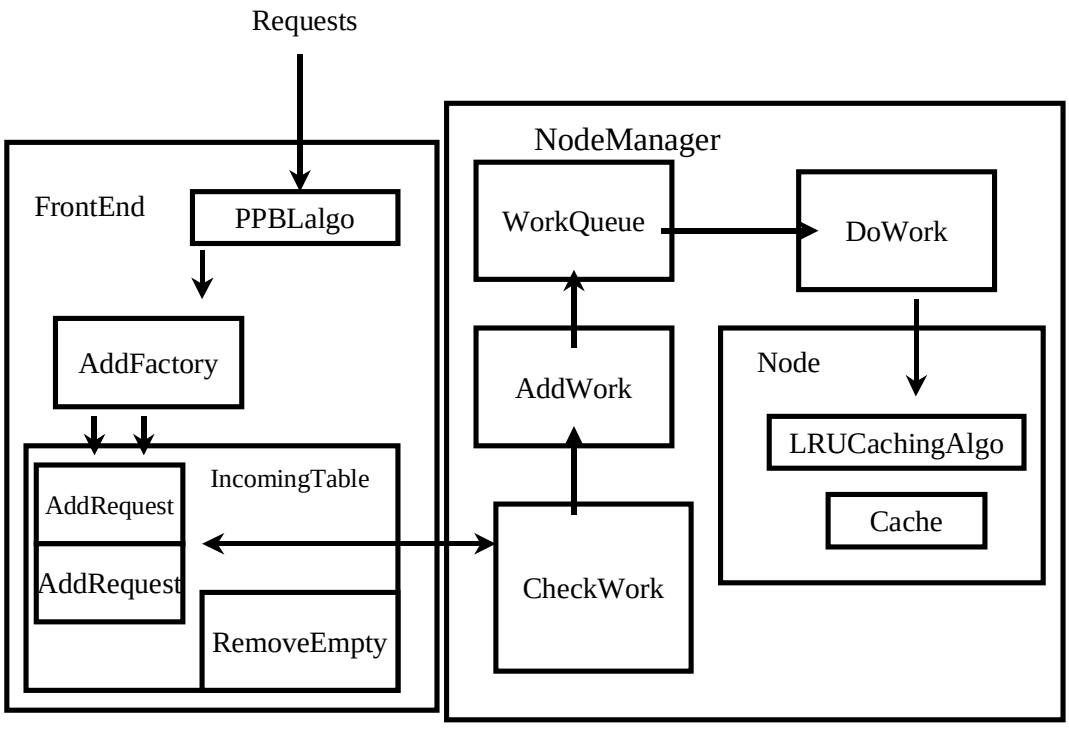

Figure 9- Parallel Pull-Based LRU Architecture

In the PPBL simulation, new `Request` objects are sent by the `FrontEnd` to the `AddFactory`. Here each request is inserted into an `AddRequest` thread, and this thread is put into the `IncomingTable`. This table will contain a list of active threads awaiting to be assigned to the appropriate node. The `IncomingTable` is periodically scanned by each `NodeManager`'s `CheckWork` thread. This thread is woken up every time a new `AddRequest` thread is inserted into the `IncomingTable`. If `CheckWork` recognises a `AddRequest` as a request for which it is responsible, it claims the request by inserting its node identifier into the `IncomingTable` entry. This will allow the `AddRequest` thread to die, as it is no
longer needed. If the `AddRequest` contains a request that is not yet assigned, then once all `CheckWork` threads have examined it, it will assign this `Request` to the least loaded (in other words, to the top of the stack) node in the `LeastLoadedQueue`. Then next time the `CheckWork` thread examines this entry
to see if it is completed or not, it will see that this `Request` has been assigned to its `Node`, and it will pass that `Request` along.

The `IncomingTable` also has a `RemoveEmpty` thread running. This thread activates after a certain number of entries have been inserted into the `IncomingTable`. This thread will order the `AddFactory` and `CheckWork` threads to lock. When all those threads have locked, the `RemoveEmpty` thread will scan through the `IncomingTable` and remove entries that have been either claimed by `CheckWork` threads or have been assigned to a `Node` and this has been recognised by the appropriate `CheckWork` thread. The `AddThread` in charge of the entry will have already died as its duties have been completed. Once the `RemoveEmpty` thread has finished with the `IncomingTable`, it wakes all the threads waiting on the `stop` lock of the `IncomingTable`, and goes back to sleep.

Once the `CheckWork` thread has claimed a `Request` from the `IncomingTable`, it passes it onto the `NodeManager`. The `NodeManager`

then inserts it into the `WorkQueue`, and updates the load on this node accordingly. The load information in the PPBL simulation is based upon the total size of `Request` objects still to be served by this `Node`. This is the total size of all requests that are in the `WorkQueue`. Each `NodeManager` also has a `DoWork` thread running. This thread is woken when new work is inserted into the `WorkQueue`. The `DoWork` thread passes the `Request` onto the `LRUalgo` thread, which is the cache replacement algorithm used in PPBL. The `StatsObject` for that document is then updated.

This PPBL simulation is somewhat simplified from the implementation put forth in [C01]. However these simplifications have not affected the dispatching algorithm of PPBL at all, but are more simplifications in the optimisations suggested in [C01] to do with synchronisation. As was suggested in [C01], we envisage that in order to build an efficient and scalable application, a good knowledge of the mechanisms involved in the middleware of the DSM system is necessary.

5 Web traces, examination of input and statistics

Trace-driven simulation is used to evaluate the performance achieved with LARD, PPBL and RR. The workload for our simulations was drawn from empirical traces, obtained from access logs at two Web servers [WL01]. The first trace is called the EPA trace in the rest of this paper. This is the access log of a day at a busy EPA WWW Server. The second trace is called the SDSC trace. This is the access log of a day at a busy San Diego Supercomputing Centre Web server. The distribution of requests in the traces can be seen in Fig 10 and 11. In each graph the x-axis ("Times Requested") represents the number of times that a certain document was requested by the trace, and the y-axis ("Total Documents") represents the total number of documents that were requested x times. In the EPA trace, a total of 242 documents were accessed more than 20 times, and in the SDSC trace, 208 documents were accessed more than 50 times.

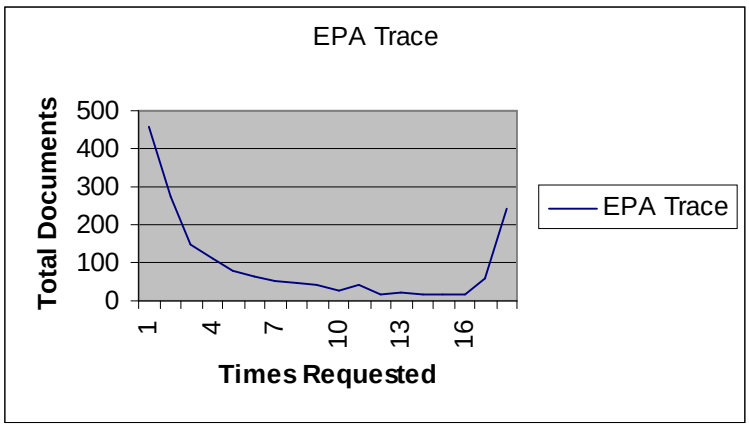

Figure 10 EPA Trace Topology

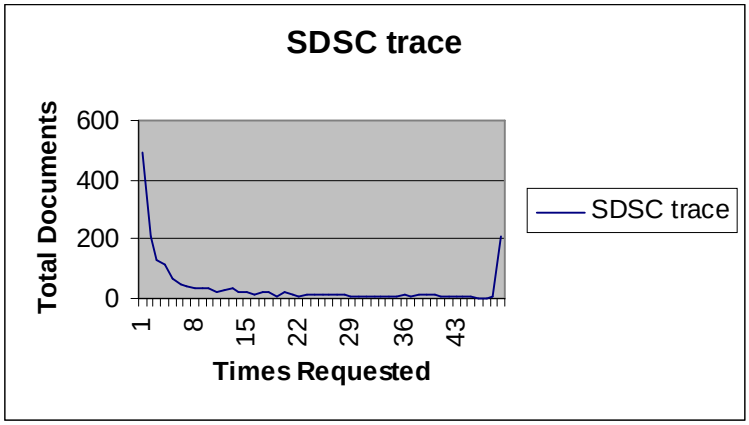

Figure 11 SDSC Trace Topology

The EPA trace had a total of 44159 requests, which accounted for a set of 4860 documents. The total size of all 4860 documents (the working set) requested in the trace was 248M. The SDSC trace had a total of 63080 requests, accounting for a set of 1864 documents. The total size of the 1864 documents was 102M.

By using two traces with different characteristics we will get a good idea of how the simulated clustered Web servers will act in general. The traces provide the time, name and size in bytes of every request. This information is transformed into a stream of `Request` objects containing the relevant name, time of the request and the size of the request, that the simulation will process. CGI scripts were removed from the trace before processing, as these dynamic requests would skew the results of our simulation. This is because they require CPU time from the Web server, and this cannot be simulated satisfactorily in our simulation. Any unique URL requests

are also removed from the trace (URLs with a "?" in them for example, which are supplying parameters for the Web server to act upon).

6 Evaluation of results

The simulations calculate the percentage of cache misses. In a Web server, these cache misses will result in a disk access. As expected, the RR cache performance deteriorates as the number of nodes increases. This is consistent with results found in other simulations [APS98]. LARD also performs as predicted in [APS98], although the performance of LARD in the EPA trace does not improve as much as expected with the increasing numnber of nodes. Overall however, the LARD simulation can be seen to behave as expected.

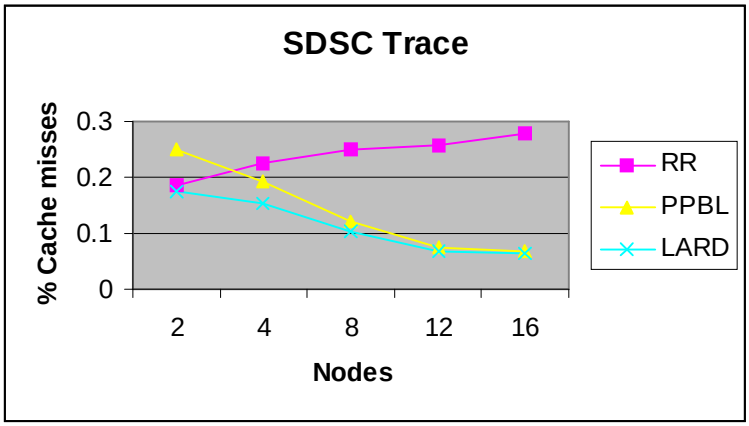

Figure 12- SDSC % Cache Misses

The PPBL simulation performs almost as well as LARD, and as the number of nodes increases the percentage of cache misses of PPBL approaches that of LARD. It should also be noted that PPBL was implemented with the sub-optimal LRU cache replacement strategy, as put forth in [C01]. It is expected that implementing Parallel Pull-Based Greedy Dual Size (PPB-GDS) will further improve the cache hit rate.

These preliminary results show that both LARD and PPBL can achieve a highly satisfactory cache hit ratio. Load balancing in the three simulations can be seen in figure 14. This graph shows the amount of requests processed per node for the EPA trace given a cluster of 8 nodes. The load balance for RR is perfect, as this is the

only requirement of this Request dispatching algorithm. LARD has reasonable load balancing, as load balancing is a requirement of the system. It can be seen however that the load balancing of PPBL is quite skewed, with node 1 in the cluster having received only 3140 requests, and node 8 6649 requests. This can be explained by the approach taken to load balancing in PPBL. While it is interesting that the load balancing decision is made by the node when it receives its request, and thus it can push that request onto another node if it feels it is overloaded, we do not have the centralised load balancing of either RR or LARD. LARD also has the added advantage of being able to limit the amount of connections in the cluster at one time, a restriction that might also benefit the performance of PPBL. However in a realistic implementation of LARD, an examination on whether the limit placed upon the total connections allowed in the cluster at one time restricts the performance of the cluster, and is therefore enforcing load balancing at the cost of throughput. The results of [APS98] would indicate that this is not the case.

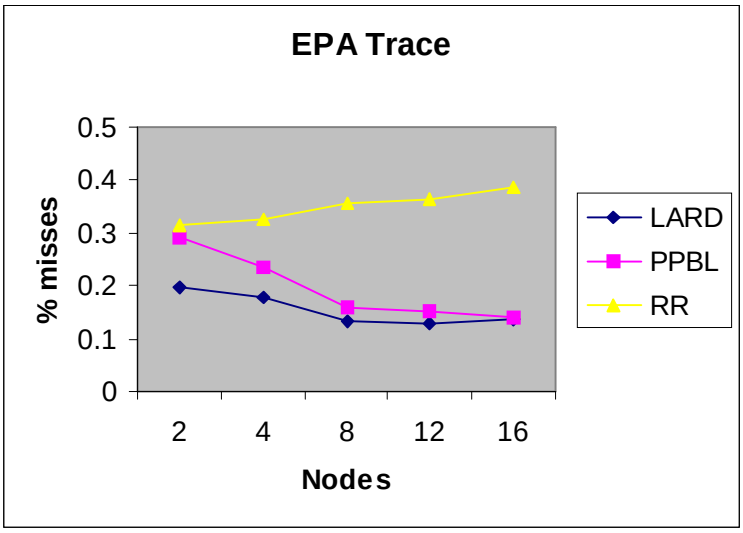

Figure 13 EPA % Cache Misses

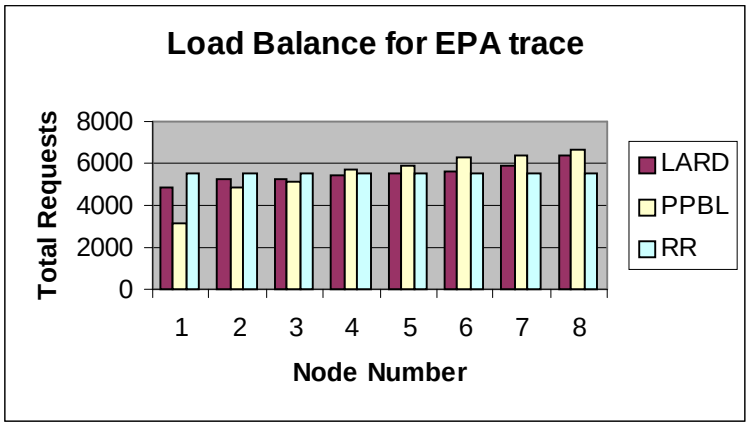

Figure 14- Load Balancing Comparisons

7 Conclusions

By simulating a clustered Web server we have learnt a great deal about the issues that govern this area. While the simulation does not indicate a great deal of difference between the PPBL and LARD strategies in terms of measured results, their approaches to the architecture of their clustered Web servers do show the differences in the two systems. While PPBL is designed with a DSM model in mind, it does introduced a great deal of complexity into the design and implementation of the clustered Web server. Lets first look at the areas of clustered Web servers that need to be focused upon when designing such a system.

An area that needs more research is dynamically created Web documents and Web Application Servers. This domain needs more research into how a clustered Web server can optimise the performance of the services it offers. One approach is to partition the nodes of the cluster into specialist service providers. For this scenario, a content-aware dispatching algorithm is crucial. It is in this area that the WRR approach to clustered Web servers falls far short of the content-aware approaches. At the moment, a WRR cluster cannot deal with a variety of services offered in a satisfactory way, unless an approach like CAP is taken, but that approach necessarily involves content-aware decisions. Another approach to optimising the performance of dynamic and specialised services is to examine how these services can be cached in a useful fashion. This approach will require more research into the

possibilities of caching dynamic services and perhaps the partitioning of these services to enhance their locality.

One area that is not exploited fully in [C01] is the possibilities that a DSM offers to a clustered Web server in terms of caching. If remote nodes can store data in their main memory and have that data accessible 50 times quicker than a local disk access, then perhaps the concept of caching in a DSM cluster needs to be re-addressed. It might be feasible to have a global cache manager, who is in charge of the contents of the caches of all nodes. This might allow the clustered Web server designer to present an aggregrate cluster cache that performs better than the current system of isolated caches, which have improved performance through the use of locality-aware decision-making in the dispatcher. One possible architecture could be using a RR approach with a cache that attempts to aggregate all the caches into one global cache, and therefore improves the cache performance on a per node basis while still gaining the benefits of WRR, which are excellent load-balancing and the lack of a TCP Handoff. Another possible improvement might be to utilise remote-node main memory as much as possible in a DSM-based clustered Web server. LARD or PPBL would have to be adjusted to achieve this, but this does not seem to be too much of a change. The issue of looking for a document remotely will only really occour when a request is swapped to another node because of excessive demand for that document causing it to be replicated (a “hot” document) or because a node in the cluster is either overloaded or too idle. In either of these cases it would be feasible to try and fetch the document from the old serving node’s cache rather than to access it from disk.

PPBL has put forth the idea of a distributed dispatcher. It has been seen in [C01] and in simulating PPBL here that increasing the number of nodes in the cluster also increases the number of members in the decision-making of the distributed dispatcher. This is attempted to be solved in [C01] through extensive use of the middleware that the clustered Web server is overlaying. However in [ASDZ00] it is argued that a centralised dispatcher is capable of up to 50,000 connections a second. That is far beyond any throughput achieved in the clustered Web servers examined in this paper. It would appear that the added complexity and performance

doubts over a distributed dispatcher would argue against this approach, especially if the Web server will employ a distributed distributor approach to the TCP Handoff problem.

In conclusion we have evaluated the Caching strategies for clustered Web servers. We have explored in details several clustered Web server architectures, such as WRR, HACC and CAP. A simulation was run to examine in more detail the effects on each individual node's caches, in the context of LARD, PPBL and RR as a benchmark for the other two algorithms. We have pinpointed several key issues in the area of clustered Web servers within the context of a DSM system. These issues need to be researched in more detail before any implementation of a clustered Web server using the Kaffemik system should be attempted.

8 Bibliography

[ASAWF95] M. Abrams, C.R. Standbridge, G. Abdulla, S.Williams, and E.A. Fox. Caching Proxies: Limitations and Potentials. Proceedings of the Fourth International Conference on the WWW, December 1995.

[APB96] E. Anderson, D. Patterson, E. Brewer. The Magicrouter, an Application of Fast Packet Interposing. Submitted to the Second Symposium of Operating System Design and Implementation, May 1996.

[APS98] M. Aron et al. Locality-Aware Request Distribution in Cluster-based Network Services. Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems October 1998.

[ASDZ00] M. Aron, D. Sanders, P. Druschel, W. Zwaenepoel. Scalable Content-Aware Request Distribution in Cluster-based Network Servers. Proceedings of the 2000 Annual Usenix Technical Conference, June 2000.

[AWCJC00] J. Andersson, S. Weber, E. Cecchet, C. Jensen, V. Cahill. Kaffemik-A Distributed JVM on a Single Address Space Architecture. Trinity College Dublin Technical Report, 2000.

[C01] E. Cecchet. Parallel Pull-Based LRU: A request Distribution Algorithm for Clustered Web Caches using a DSM for Memory Mapped Networks.

[CC01] E. Casalicchio, M. Colajanni. A Client-Aware Dispatching Algorithm for Web Clusters Providing Multiple Services. Proceedings of the Tenth International Conference on the WWW, May 2001.

[CI97] P. Cao, S. Irani. Cost-Aware WWW proxy caching algorithms. Proceedings of the USENIX Symposium on Internet Technologies and Systems, December 1997.

[CKP98] E. Cecchet, P. Koch, X. Rousset de Pina. Global Management of Coherent Shared Memory on an SCI cluster. Proceedings of the First International Conference on SCI-based Technology and Research, September 1998.

[CRS99] A. Cohen, S. Rangarajan, H. Slye. On the Performance of TCP Splicing for URL-Aware Redirection. Proceedings of the Second USENIX Symposium on Internet Technologies and Systems, October 1999.

[D97] O. Damani et al. Techniques for Hosting a Service on a Cluster of Machines. Proceedings of the Sixth International Conference on the WWW, April 1997.

[F96] M. Feeley. Global Memory Management for Workstation Networks. PhD Thesis, University of Washington, 1996.

[F97] A. Fox et al. Cluster-Based Scalable Network Services. Proceedings of the 16th ACM Symposium on Operating System Principles, October 1997.

[H99] G. Hunt et al. Network Dispatcher: a Connection Router for Scalable Internet Services. Computer Networks and ISDN Systems, September 1999.

[IEEE93] IEEE. IEEE Std 1596-1992: IEEE Standard for Scalable Coherent Interface. IEEE, August 1993.

[KHCP99] P. Koch, J. Hansen, E. Cecchet, X. Rousset de Pina. SCIOS: An SCI-based Software Distributed Shared Memory. Proceedings of the First Workshop on Software Distributed Shared Memory, 1999.

[SGR00] T.Schroeder, S. Goddard, B. Ramamurthy, Scalable Web Server Clustering Technologies. IEEE Network, May/June 2000.

[TRS97] I. Tatarinov, A. Rousskov, V. Soloviev. Static Caching in Web Servers. Proceedings of the Sixth International Conference on Computer Communications and Networks, IC3N 1997.

[WL01] Traces obtained from http://www.web-caching.com/traces-logs.html.

[ZBCS99] X. Zhang, M. Barrientos, J. Chen, M. Seltzer. HACC: An Architecture for Cluster-Based Web Servers. Proceedings of the Third USENIX Windows NT Symposium. July 1999.