Scalability Issues in Cluster Web Servers

Arkaitz Bitorika

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

2002

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.

Signed: ______________________

Arkaitz Bitorika

16th September 2002

Permission to lend and/or copy

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

Signed: ______________________

Arkaitz Bitorika

16th September 2002

Contents

1 Introduction **7**

2 State of the Art Overview **10**

2.1 Web Serving Overview 10
2.2 Cluster architecture 12
2.2.1 Layer 4 switching with layer 2 forwarding 12
2.2.2 Layer 4 switching with layer 3 forwarding 13
2.2.3 Layer 7 switching 14
2.3 Server selection / request distribution 17
2.3.1 DNS Round Robin 17
2.3.2 Weighted Round Robin (WRR) 18
2.3.3 Weighted Least Connections 19
2.3.4 LARD 19
2.3.5 WARD 24
2.3.6 PPBL 25
2.4 Web caching replacement policies 26
2.4.1 Least Recently Used (LRU) 27
2.4.2 Least Frequently Used (LFU) 28
2.4.3 Hyper-G 28
2.4.4 Pitkow-Recker 29
2.4.5 Hybrid 29
2.4.6 Lowest Relative Value (LRV) 29
2.4.7 GreedyDual-Size 30

3 Design and implementation **31**

3.1 Architecture 32
3.2 Inter-process communication 35
3.2.1 RMI protocol 35
3.2.2 Socket-based application-specific protocol 36
3.3 Concurrency model 37

3.3.1 Thread-per-connection model . . . 37
3.3.2 Event-driven model with multiplexed I/O . . . 38
3.4 Front-end . . . 41
3.4.1 Multi-threaded socket listener . . . 42
3.4.2 Reactor-based socket listener . . . 44
3.4.3 Dispatcher . . . 46
3.5 Cache Manager . . . 47
3.6 Node Manager . . . 48
3.6.1 Caching . . . 50

4 Evaluation **52**
4.1 Test setup and methodology . . . 52
4.1.1 WebStone . . . 53
4.2 Traces . . . 54
4.3 Evaluation results . . . 55

5 Conclusions and future work **61**

List of Figures

2.1 Example URL division. 11
2.2 Web server address resolution steps. 11
2.3 Layer 4/2 cluster architecture 13
2.4 Layer 4/3 cluster architecture 14
2.5 TCP gateway operation 15
2.6 Architecture of split connection application layer dispatchers. The proxy or dispatcher keeps open TCP connections both with the client and with the server. 16
2.7 Architecture of a dispatcher or layer 7 proxy using TCP splice. The request data that moves between client and server is kept at the kernel level. 16
2.8 Traffic flow in TCP gateway versus TCP hand-off 17
2.9 LARD operation 19
2.10 Pseudo-code for the basic LARD request distribution algorithm. 20
2.11 LARD with Replication 21
2.12 Centralised dispatcher and distributor 23
2.13 Scalable LARD cluster architecture 23

3.1 Common Web cluster deployment scenario. 32
3.2 Logical view of the components in a running cluster. 33
3.3 Physical view of the components in a running cluster. 34
3.4 Structure of the messages sent from the front-end to the node managers. . 37
3.5 Thread-per-connection model. 38
3.6 Single threaded event-driven model with multiplexed I/O. 39
3.7 Class diagram of main participants in the Reactor pattern. 40
3.8 Multi-threaded listener class diagram. 42
3.9 Sequence diagram for RequestHandler thread. 43
3.10 ReactorListener class diagram. 45
3.11 IDispatcher interface class hierarchy. 46
3.12 CacheManager class diagram. 48
3.13 Node manager class diagram. 49
3.14 Class hierarchy for *IresponseCache*. 50

4.1 Connections per second with dynamic traffic. 56
4.2 Connections per second with static traffic. 57
4.3 Throughput with dynamic traffic. 58
4.4 Throughput with static traffic. 59
4.5 Average response times for dynamic traffic. 59
4.6 Average response times for static traffic. 60

Chapter 1

Introduction

The World Wide Web is probably the most popular application running on the Internet, and the traffic generated by the ever growing number of users on the Web forces Web sites to implement scalable Web server architectures that will be able to handle current loads as well as grow to meet future performance requirements.

Cluster Web servers are a popular solution for building scalable Web serving platforms, they offer a cost-effective alternative to high-end servers and can be built from relatively economical hardware and software. State-of-the-art commercial Web clustering products employ a specialized front-end node that is a single point of contact for Web clients and distributes HTTP requests to the back-end nodes in the clusters. Back-end nodes are typically standard Web servers than can be built with common of-the-shelf (COTS) hardware and run a Unix-like or Windows operating system. The front-end node is normally a device that integrates proprietary hardware and software from a commercial vendor.

Web clustering front-ends can be broadly divided in layer 4 and layer 7 switches or dispatchers. Layer 4 refers to the OSI network protocol layer 4, TCP in TCP/IP networks, and means that the front-ends can only use TCP information for taking request distribution requests. In layer 7 switches the front-ends are application layer information aware, the application being HTTP in the case of Web servers. Access to HTTP information such as the request URL, file type or cookies sent by the Web browser allows for a more effective and flexible configuration of the cluster. For example, layer 7 front-ends can dispatch all requests for a certain kind of file type to a specific server, or when used with Web application servers, send all the request with the same session cookie to the back-end server that has the relevant session data in main memory.

However, state-of-the-art Web clustering solutions achieve only limited scalability , and offer a solution based on embedded hardware and software that does not integrate well with, and take advantage of, existing Web server and application software. The scalability offered by dedicated front-end nodes is limited, and they do not necessarily

integrate with the back-end nodes' software as they have to offer a solution that is as "standalone" and "plug and play" as possible. Most clustering products don't work as an integrated whole, they are implemented as separate front-ends or Web switches that "clusterise" a set of back-end Web server nodes.

The requirement for Web cluster front-ends to be transparent to the back-end nodes has important scalability implications when layer 7 switching is used. The front-end has to accept TCP connections and read their HTTP request data, tasks that were previously performed by the back-end Web server. This extra work can make the front-end become a bottleneck, and with front-ends that are transparent to the back-end servers the HTTP request decoding is done once more by the back-end Web server as well.

In this dissertation a Web cluster architecture is designed, implemented and evaluated. We believe that with the use of COTS hardware and integrated front-end and back-end software better scalability and flexibility can be achieved for Web clusters. Furthermore, the clustering software is all implemented in Java, with the goal of evaluating Java as a platform for the implementation of high-performance network servers. The Web cluster prototype described here is built out of a set of distributed components that collaborate to achieve good scalability as a Web server. As all the components have been designed from scratch to integrate with each other and work well together, the scalability problems associated with state-of-the-art content-aware commercial clusters are avoided.

The prototype Web cluster described in this dissertation will be evaluated using publicly available Web benchmarking tools and Web access-log traces. Different workloads will be applied to the cluster as well as to a standalone Java Web server to compare their performance and scalability. Web clustering research has frequently assumed that the Web traffic is mostly static, that is, mainly composed of requests for static files. However, more and more Web sites implement dynamic Web applications and there is a tendency toward completely dynamic sites. Thus, apart from standard static traffic dynamic Web traffic is also simulated and evaluated in our tests.

The evaluation results show that a differentiation between static and dynamic requests is necessary for better understanding of Web server scalability issues. When handling static traffic, it is easy for a single node Web server to saturate an Ethernet 100 Mbit/second connection, and I/O throughput is the main factor. However, scalability of Web servers that handle dynamic traffic is much improved from using a clustering approach. This is due to the fact that when serving dynamic requests I/O performance is not the main scalability factor and CPU processing becomes very important as well. Furthermore, issues involved in implementing scalable network servers in Java are discussed as well, focusing specifically on how the new non-blocking I/O facilities affect the architecture and scalability of server applications.

This dissertation is organised as follows: chapter two contains a state of the art

overview of Web clustering, chapter three discusses the design and implementation of an scalable Web cluster in Java, chapter four evaluates the performance and scalability of the cluster prototype and chapter five presents conclusions and future work.

Chapter 2

State of the Art Overview

2.1 Web Serving Overview

URLs are the most user-visible component of how the World Wide Web (Web) works. When a user wants to retrieve a Web page, all the necessary information for contacting the right Web server and making the appropriate request for the desired page is contained in a URL that the user can type in the address bar of the browser. URLs are composed by several parts as seen in figure 2.1, they are:

Protocol The protocol specifies which application layer (OSI layer 7) protocol the browser will use to communicate with the server. In the case of Web connections, the protocol will normally be http for traditional clear-text requests or https for secure encrypted requests. These two are the standard Web protocols, but current browsers usually understand others like ftp or mailto, even if they are not strictly part of the Web.

Host The host name will specify which Web server a request is directed to. It is a human-understandable name that will be translated to an IP address by the DNS naming system used in the Internet. Once the browser obtains the IP address that the host name belongs to using DNS, it can directly address the Web server host.

Port The port component of the URL is an optional part required by the underlying network protocol layer, TCP. Even if the Web client will always use a port number when connecting to the server, the port is optionally in the URL, as different protocols have well-known ports that are chosen by default if no specific port number is given. For example, HTTP uses 80 as its well-known port number.

File path It contains the path to a Web page or file in the Web servers file system, expressed in a form relative the document root of the Web server, not to the operating-system root of the server's file system. The format for the path uses Unix conven-

tions for separators. With the popularity of dynamic sites and Web applications, the path frequently doesn't specify the location of a physical file that exists on the servers file-system, instead it serves more as a Remote Procedure Call that the Web server will use to decide which part of the Web application logic to execute.

http://www.example.com:80/index.html

protocol://host:port/filePath

Figure 2.1: Example URL division.

The initial URL that the Web browser user types in an address bar will eventually be mapped to a set of blocks in a file-system that a Web server process will read or a set of instructions that will generate a dynamic response.

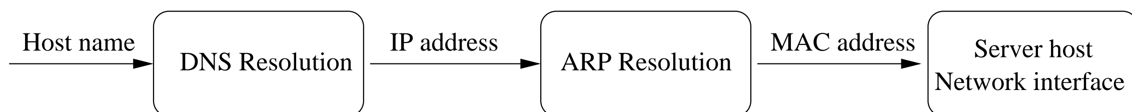


Figure 2.2: Web server address resolution steps.

There are various steps in this process assuming that the final server is connected to an Ethernet network (Figure 2.2):

DNS Resolution The Domain Name System (DNS) is the standard mechanism on the Internet for resolving a given host or domain name to an unique IP adress. The IP address identifies one physical machine connected to the Internet.

IP Routing When the Web browser obtains the IP address from the name resolution subsystem, it sends an HTTP request to that IP address, by sending a set of IP packets with the IP address of the host. These IP packets go through potentially several hops between Internet routers until they reach the sub-network that is configured as including the host with the IP address.

ARP Resolution When the request IP packets reach the last router before the host, the router will resolve the IP address of the host to an Ethernet MAC address. A MAC address is an unique identifier of the network card installed on the host. The mapping between IP and MAC addresses translates allows the Ethernet switch to send the request packets to the right network interface connected to it.

Once the request IP packets arrive to the Web server host, they get reassembled to build a HTTP request, and the server maps the URL to a file in the local file-system. The Web

server will use then the standard operating system facilities to read the disk blocks needed for obtaining the contents of the file.

This process has several steps in which a resolution of a mapping between different naming or addressing schemes is performed: from host name to IP address, from IP address to MAC address and from requested URL to file-system block. These points in the process of handling a HTTP request are where Web clustering techniques can be introduced. For example, DNS round-robin will resolve a given host name to one of several IP addresses available, or a Web server can cache files in main memory instead of loading them from the file-system for each request. Several examples of cluster architectures are discussed in the next section.

2.2 Cluster architecture

We will use a classification of different Web server clustering technologies that has been proposed explicitly by Steve Goddard and Trevor Schroeder [32, 33], but is common as well in the research literature. Most of the Web server clustering technologies are transparent to the client browsers, the browsers are at no time aware of the existence of clustering on the server side. However, not all clustering technologies are transparent to the web server software. While still being transparent to the clients, some clustering solutions are visible to the Web servers, and need specialised software at different levels in the server system.

2.2.1 Layer 4 switching with layer 2 forwarding

In L4/2 clustering, the dispatcher and the servers all share a common cluster network address (referred as "Virtual IP address" by some vendors). All incoming traffic for the cluster address is routed to a dispatcher, using static ARP entries, routing rules, or some other mechanism. All the servers have a primary unique IP address, but share the cluster address with the rest of the servers and the dispatcher, which has the cluster address as its main IP address.

When a packet arrives, the dispatcher determines whether it belongs to a currently established TCP connection or is a new connection. If it is a new connection, the dispatcher chooses one target server to satisfy the request according to its request distribution policy and saves the mapping between the connection and the server in its memory. Then the MAC (physical layer or layer 2) address of the frame is rewritten and the packet is sent to the chosen server.

The server receives the packet and it accepts it as the packet has the address of the cluster, which all the servers have been configured to answer to. Replies go back through

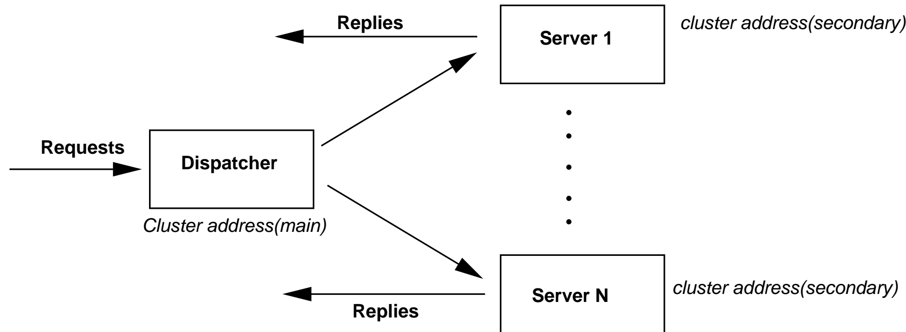


Figure 2.3: Layer 4/2 cluster architecture

the servers' default gateway, instead of going through the dispatcher, avoiding the bottleneck of passing all the traffic through the dispatcher. When the TCP connection gets drop, the dispatcher will delete the associated information from its memory.

This technique is very simple while still providing high throughput, as only the incoming traffic is processed by the dispatcher. In typical Web traffic, replies account for most of the data transmitted, and with L4/2 clustering the reply data is sent straight from the server to the client. Moreover, as only layer 2 information is rewritten at the dispatcher, there is no need to recompute TCP/IP checksums.

The drawback is that all the nodes have to be in the same switch as the dispatcher, due to its use of layer 2 addressing, which is not usually a problem as clustered servers are usually interconnected by high-speed LANs, and it could be possible to use proprietary solutions like Cisco switch-to-switch interconnects to propagate MAC address between different switches.

Research prototypes of L4/2 clustering are ONE-IP [24] from Bell Labs and LSMAC [20] from the University of Nebraska. Most of the commercial load balancing solutions provide this kind of clustering as an option.

2.2.2 Layer 4 switching with layer 3 forwarding

It is also known as "Load Sharing Using Network Address Translation (LSNAT)", and it is an Internet standard detailed in RFC 2391[30]. Examples of commercial implementations are Cisco's LocalDirector 400 series [5], Foundry Networks' ServerIron web switches[11] and Nortel's Alteon ACEdirector series [23]. A research prototype using it is MagicRouter from Berkeley [22].

In L4/3 each server in the cluster has a unique IP address, and the dispatcher is the only one that has the address assigned as the cluster address. When a packet arrives the dispatcher, it rewrites the destination IP address to be the address of the chosen server out of the cluster, recomputes necessary checksums, and sends it back to the network. When the packets are TCP connection initiations, the dispatcher will choose one of the cluster

servers using a request distribution algorithm, keeping the association between layer 4 connection and the server for all future packets part of the connection.

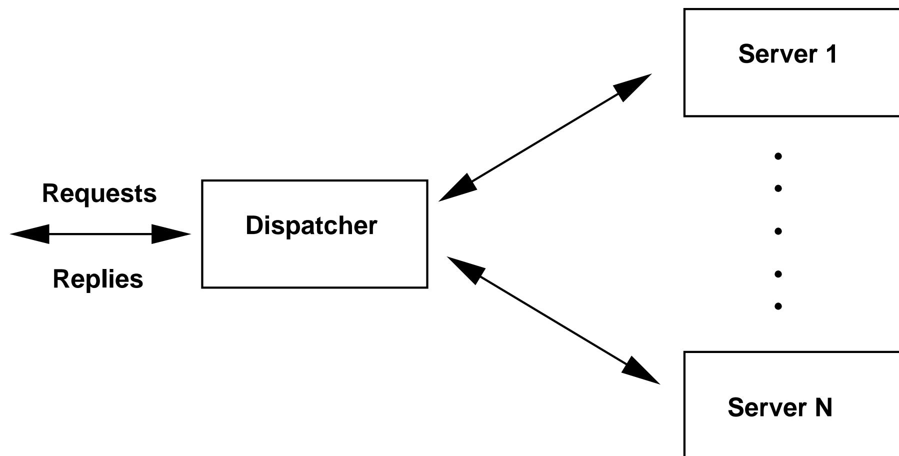


Figure 2.4: Layer 4/3 cluster architecture

The server processes the packet and sends the response back to the dispatcher. This last step is necessary, as without changes in the network protocol, the server operating system, or device drivers, the packet must be sent to the dispatcher, because the client expects a reply from it. When receiving the response, the dispatcher changes the source address from that of the responding server to the cluster IP address, recomputes checksums, and sends the packet to the client.

It is well known that a L4/3 cluster will perform worse than a L4/2 architecture, due to the workload imposed upon the dispatcher by L4/3. As all the traffic has to go through it and it has to compute the checksums for each packet, the dispatcher becomes quickly a bottleneck .

2.2.3 Layer 7 switching

Also known as content-based switching or content-aware request distribution, it operates at the application layer (L7) of the OSI protocol stack. Commercial implementations that have this functionality are Nortel's Alteon ContentDirector [23], Foundry Networks' ServerIron family [11]and Cisco's CSS 11000 switches [5].

In an L7 cluster the dispatcher is a single point of contact for clients, similarly to the layer 4 switching architectures. However, the dispatcher does not merely pass packets depending on a load balancing algorithm or which TCP connection they are from. Instead, the dispatcher will accept the connection and choose an appropriate server based on application level information sent by the client in the request.

There are two main approaches to the handling of the connection once the target back-end server has been chosen, TCP gateway and TCP hand-off.

TCP Gateway

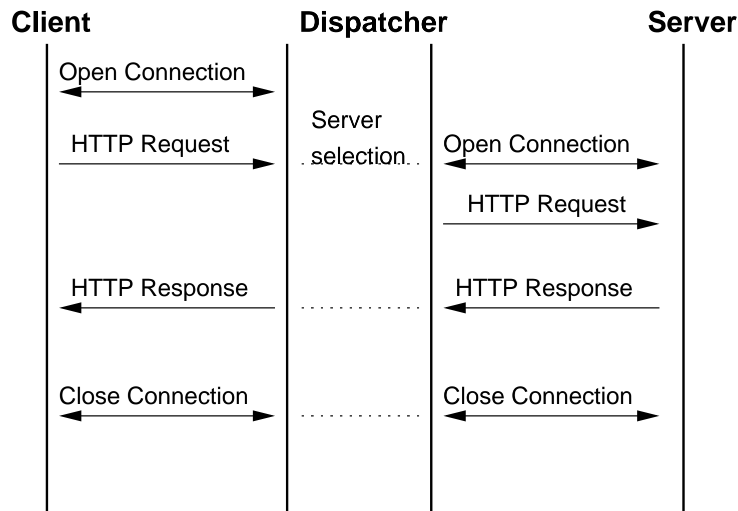


Figure 2.5: TCP gateway operation

In this case, once the dispatcher has decided which back-end server will handle the request, a new TCP connection is established between the dispatcher and the server, and the dispatcher acts basically as a TCP gateway, relaying the data between the server and the client. This approach has similar problems to L4/3 switching, all data has to go through two connections and TCP/IP protocol stacks, the first one between the client and the dispatcher and the second one between the dispatcher and the back-end server.There are various optimisation techniques used with this architecture, connection pooling and TCP splicing.

Connection Pooling

Creation of TCP connections is a time and resource consuming operation, to avoid this time the dispatcher can “pre-fork” connections to the servers, keeping a pool of open TCP connections that it can reuse. This way, once the target server has been chosen the dispatcher simply picks an idle connection from the pool instead of establishing a new connection with the server, avoiding the overhead of initiating a TCP connection for every new HTTP request. A research prototype of this design is used for Content-based Routing[9].

TCP Splicing

Typically, proxies and load-balancing dispatchers that work at the application layer use an architecture called *split connection*, where there is a connection between the client and the proxy/dispatcher and a different connection exists between the proxy and the server.

This scheme illustrated in figure 2.6 has important performance problems and violates end-to-end semantics of TCP.

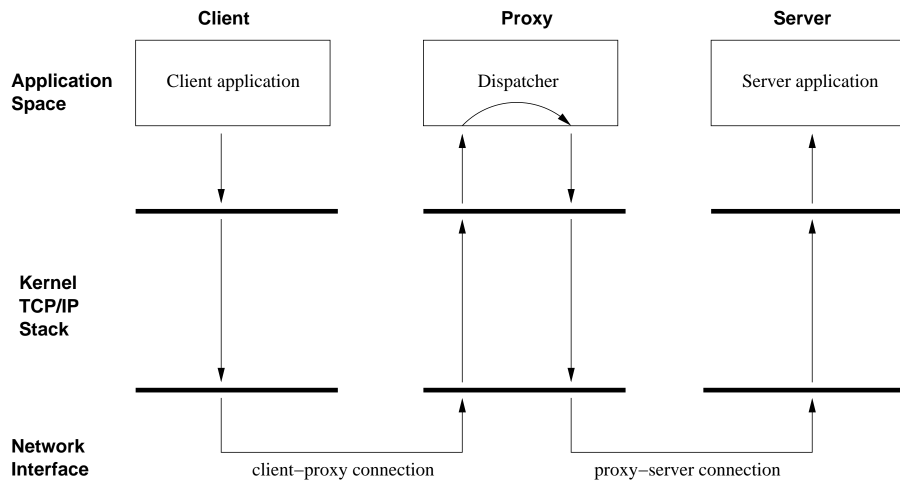


Figure 2.6: Architecture of split connection application layer dispatchers. The proxy or dispatcher keeps open TCP connections both with the client and with the server.

TCP Splice has been proposed by David Maltz [35] as an improved design for TCP gateways. The basic idea behind TCP Splicing is shown in figure 2.7, the packets that are subject to being forwarded can be modified as they are received without passing them up and down through the whole TCP/IP stack. Once the dispatcher has set up the splice for a given connection, all the packets that are part of this connection go through the splice, without performing all the expensive data-copying operations caused by the transit through the protocol stack.

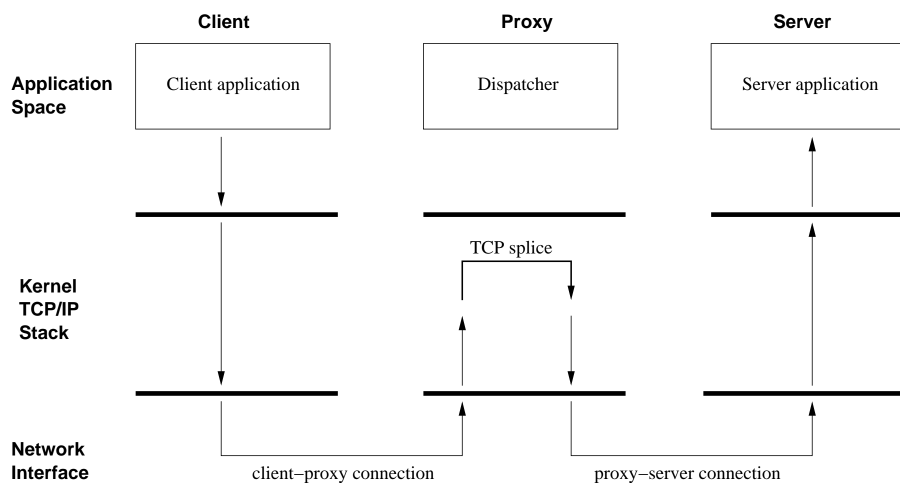


Figure 2.7: Architecture of a dispatcher or layer 7 proxy using TCP splice. The request data that moves between client and server is kept at the kernel level.

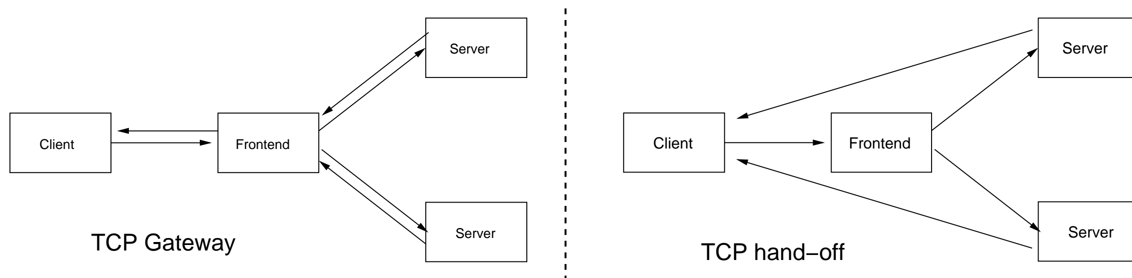


Figure 2.8: Traffic flow in TCP gateway versus TCP hand-off

TCP Hand-off

This approach, illustrated by figure 2.8allows the dispatcher to give or "hand off" a TCP connection to a particular server once it has decided which one of the servers is the right one to answer the request. In layer 7 switching clusters, dispatchers need to be able to fully establish a TCP connection with the clients to receive the application layer (HTTP in this case) request data that will allow them to make a choice regarding which back-end server will deal with the request. Once the decision has been made, the dispatcher has to be able to "re-establish" the connection on the back-end server, recreating the required connection state. This protocol is extensively discussed in Thomas Larkin's dissertation [18] and is used in layer 7 designs like *Content-Aware Request Distribution* [4] or *Locality-aware Request Distribution* [17]. Resonate [29] commercialises a proprietary solution that uses this approach under the name of *TCP Connection Hop* in their Central Dispatcher product [29].

2.3 Server selection / request distribution

2.3.1 DNS Round Robin

In a Server selection system that uses DNS, the DNS server will resolve the name of the service or domain to a different IP address using different algorithms such as load average of a simple round-robin. This mechanism is specified in RFC 1974 [31]. DNS is the naming system used in the Internet, as such it supports massive scalability by making use of hierarchical organisation of naming information and caching at several levels of the naming system.

DNS-based server selection techniques base their effectiveness on setting a very low (down to 0) value for the TTL (Time To Live) field of their responses, minimising caching of the name resolutions, which allows the implementation of round-robin or other load balancing algorithms, as well as more complex solutions that try to resolve to IP addresses that will give better service based on the physical location of the client. If clients contact the DNS server for each request, the DNS server that is in front of a web cluster can

control the back-end server to which each request gets sent.

The problem with this approach is that all the elements involved in resolving a name to an IP address, from the browser to the root DNS servers, use caching for improving performance and response times to users. Between DNS servers, the validity of a piece of information is given by a TTL field that specifies how long that information can be cached before refreshing it from the original or "authoritative" source, and web browsers are known to cache resolution information for about 15 minutes. This caching at several layers is what makes DNS so scalable.

It has been shown in [10] that DNS-based server selection schemes actually have a negative impact on client-perceived web access latency. Without caching, the client should always need to contact the DNS server before each request, increasing name resolution overhead by up to two orders of magnitude. This problem is aggravated by the delays caused by DNS queries needed for embedded objects in HTML pages such as images.

2.3.2 Weighted Round Robin (WRR)

Weighted Round Robin (WRR) is the most common distribution algorithm found in commercial web load-balancing solutions, and although commercial vendors frequently support other more complex server selection schemes such as URL-aware selection, WRR and its variants are still very popular and widely used, thanks to their simplicity and scalability.

In this server selection scheme the front-enback-end dispatcher is usually a Layer 4 switch that distributes request to back-end servers using only load-balancing requirements. As its name implies, it is a variant of Round-Robin that instead of "blindly" dispatching requests iteratively to the available back-end servers, has a weighting introduced for each server, normally calculated as a function of the CPU, disk and/or memory use. The weight of each server in the load-balancing algorithm is reevaluated periodically.

The main benefits of WRR are that it obtains low idle times in the back-end servers and achieves a very well balanced load distribution. However, the main problem of WRR is that it show a high cache miss ratio. In web servers, being able to cache the working set (the set of web objects that the server accepts requests for) is critical, as the throughput of the current disk devices is much lower than the expected throughput of network servers. For web server clusters that have big working sets this supposes a problem, as each individual node main memory cache has to fit the entire working set, which is frequently impossible. When the node caches are not able to hold all their working set, the throughput is low, as the cluster is not taking advantage of its aggregated cache size. WRR scales well to high-traffic sites but doesn't provide greatly improved response rates when the working set bigger than the node caches.

Different distribution schemes based on Layer 7 switching try to overcome this problem by distributing requests to nodes that are more likely to already have the requested object in the cache, effectively partitioning the working set and making it possible for individual nodes to keep their respective partitions in cache. One example of this design is Locality-Aware Request Distribution [17].

2.3.3 Weighted Least Connections

Weighted Least Connections (WLC), is a common alternative to WRR as a request distribution algorithm. It is a weighted variant of Least Connections (LC), which bases its distribution decisions on the number of open TCP connections each back-end server has open. The rationale behind this idea is that if the server has several open connections, it means that it is busy servicing requests, and the more open connections it has, the less new requests it should receive.

The weighted version of LL, WLC, goes one step further and includes a weight in the distribution algorithm. This weight is calculated and updated the same way as with WRR, using variables such as disk, CPU or network resources usage. This algorithm is available for several load-balancing solutions, in software Linux Virtual Server [21] is an example and in hardware Cisco Catalyst web switches [5] provide it as well.

2.3.4 LARD

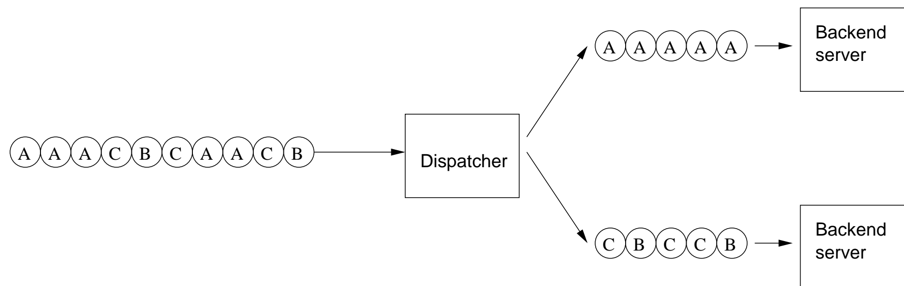


Figure 2.9: LARD operation

Locality-aware request distribution (LARD) is a form of content-based request distribution described in [17]. It uses Layer 7/HTTP information. The main advantages of content-based request distribution compared to schemes like WRR or WLC are:

1. Better throughput thanks to a lower number of main memory cache misses in the back-end servers.
2. Ability to partition the working set over the different back-end servers, having more available disk space.

3. Ability to use specialised back-end nodes for services like multimedia streaming or dynamic web applications.

LARD focuses on obtaining the first of the advantages cited above, improved cache hit rates. In figure 2.9 the operation of a basic locality-aware dispatcher is shown, by distributing requests in this way, there is the risk that the load on the back-end servers becomes unbalanced, resulting in worse performance than with simpler round-robin algorithms. Thus, LARD has to provide a solution that simultaneously achieves load-balancing and high cache hit rates on the back-end servers.

Basic LARD

The basic version of LARD always assigns a single back-end server to handle a given target/URL, effectively assuming that a the requests for a single URL can't exceed the capacity of a single server.

Figure 2.10 shows the pseudo-code for basic LARD. The dispatcher maintains a one-to-one mapping of requested URLs to back-end servers in the *server* hash-table. When a given URL is requested for the first time, the least loaded back-end server is assigned to it, and the decision is recorded. In subsequent requests for the same URL, the same server will be assigned unless the server is overloaded, in which case the request will be assigned to the least loaded node of the cluster.

The number of open connections are used to calculate the load on the back-end servers. An overloaded server will have a higher number of open connections as it will not be able to handle them quickly, while a lightly loaded node will have few or none active connections. This factor is the easiest way for the dispatcher to get load information from the back-end servers without using explicit communication, but other systems that involve communication could be used as well.

```
while (true)
    request = fetch next request
    if (server[request] is null)
        node, server[request] = least loaded node
    else:
        node = server[request]
        if ((node.load > T_high and exists node with load < T_low) or
(node.load >= 2 * T_high))
            node, server[request] = least loaded node
    send request to node
```

Figure 2.10: Pseudo-code for the basic LARD request distribution algorithm.

Basic LARD considers only locality when partitioning the working set. It keeps this strategy unless a "significant load imbalance" is detected, in which case it reassigns the URL to a different back-end server. The algorithm to reassign an URL works as follows: T_{low} is defined as the number of active connections (load) below which a back-end server will have idle resources and T_{high} as the load above which the server's response times will increment substantially. There are two conditions that can trigger a reassignment of the URL to the least loaded node:

- When a server has a load larger than T_{high} wile another server has a load less than T_{low}.
- When the server reaches a load of $2T_{high}$, even if no server has a load lower than T_{low}.

The two conditions try to ensure that the reassignment only happens when the load difference is substantial enough. To prevent the case in which the load on all servers rises to $2T_{high}$,causing LARD to behave like WRR, the total number of connections handed to the back-end servers is limited by the value $S = (n - 1) * T_{high} + T_{low} - 1$, where n is the number of back-end servers.

LARD with Replication

```
while (true)
    request = fetch next request
    if |serverSet[request]| is 0:
        node, serverSet[request] = least loaded node
    else:
        node = least loaded node in serverSet[request]
        mostLoaded = most loaded node in serverSet[request]
        if ((node.load > Thigh and exists node with load < Tlow) or
(node.load >= 2 * Thigh))
            p = least loaded node
            add p to serverSet[request]
            node = p
        if ((|serverSet[request]| > 1)and (time() -
serverSet[request].lastMod > K))
            remove mostLoaded from serverSet[request]
    send request to node
    if (serverSet[request] changed in this iteration)
        serverSet[request].lastMod = time()
```

Figure 2.11: LARD with Replication

There is a potential problem with the basic LARD strategy: only one backend server can handle each URL at any given time. If one URL or document receives too many

requests for a backend, it is possible that the backend gets overloaded, degrading performance considerably. The solution is to allow several backend servers to handle a single URL in a round-robin fashion. This is called “replication” in the LARD design, as the objects are replicated in more than one server.

Figure 2.11 contains pseudo-code for this approach. The main difference is that the dispatcher maintains a set of backend servers that handle each URL. Normally, the requests are assigned to the least loaded node in the server set. In the case of a load imbalance occurring, the least loaded node from the cluster will handle the request and will be added to the server set for this URL. To prevent the server set from growing until including all the backend servers for each URL, the dispatcher removes the most loaded server from the server set if the set hasn’t been modified for K seconds.

This design has several advantages: it doesn’t require any extra communication between the dispatcher and the backend servers, it is independent of the local replacement policy used by the caches of the backend servers and as it doesn’t use any complex state information in the dispatcher the recovery in case of failure is very simple.

In the trace-driven simulations run on LARD [17], it is shown that it performs better than state-of-the-art WRR strategies, improving performance by a factor of two in the case of working sets that don’t fit in a single backend server’s main memory cache. Interestingly, simulations show that outperforms global memory system clusters that use WRR as well, even with the advantages of being able to manage a global shared main memory cache.

Scalable LARD

The standard cluster architecture of the proposed LARD implementation uses a centralised dispatcher or front-end switch and the TCP hand-off technique. TCP hand-off is indeed more scalable than other TCP gatewaying designs such as TCP splicing, but empirical experiments have shown that peak throughput of TCP hand-off is limited to about 3500 connections per second with real trace workloads and doesn’t scale well beyond four cluster nodes [4].

The centralised nature of the content-aware request distribution strategies (e.g. LARD) makes it difficult to obtain scalability by using multiple front-end switches, which furthermore would introduce load-balancing concerns. The approach taken [4] separates the functionality of the front-end node in two components, a distributor and a dispatcher, as shown by figure 2.12. The dispatcher is in charge of deciding which backend server will handle each request, while the distributor is the component that actually performs the action of passing or “handing off” the request to the server. With this division, it can be appreciated that while the dispatcher needs to be centralised to be able to implement LARD, the distributor can be distributed over all the backend servers as it is completely

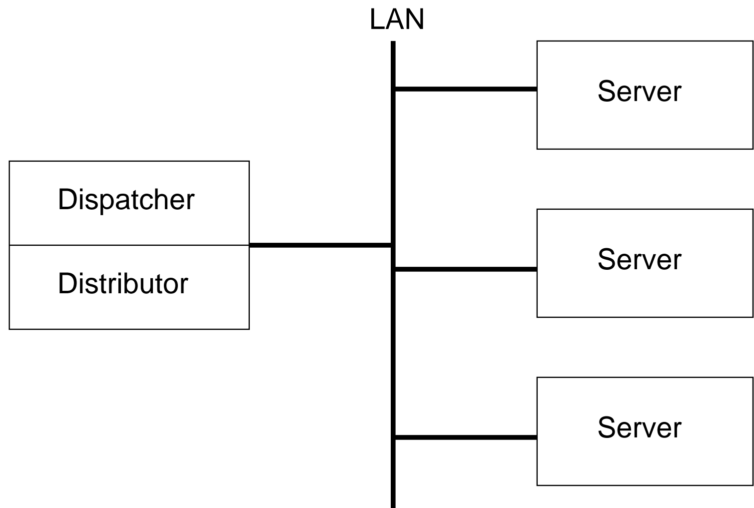


Figure 2.12: Centralised dispatcher and distributor

independent of any centralised control, while still being able to keep a centralised content-aware dispatcher as shown in figure 2.13.

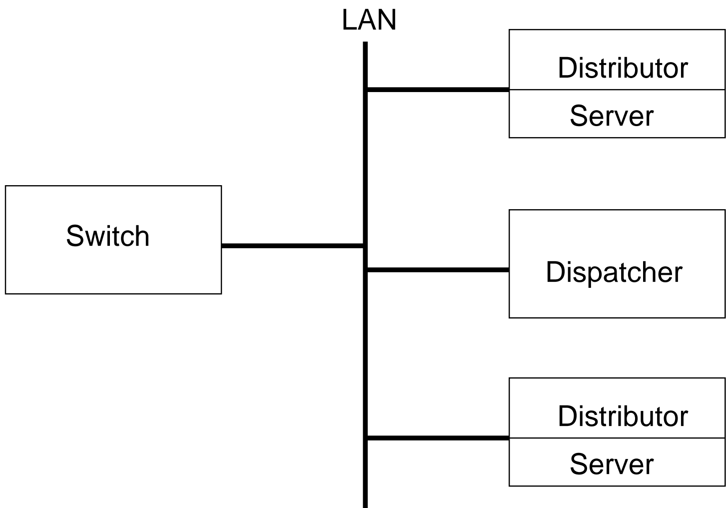


Figure 2.13: Scalable LARD cluster architecture

To be able to use distributed distributors, this architecture uses a fast layer 4 switch that is the single point of contacts for clients, working the same way as other Layer 4 cluster architectures. The main difference is that once the requests hit the distributors, the dispatcher will be contacted and the distributor that received the request will hand off the request to the backend server that the dispatcher considers adequate. As the switch does not perform any content-aware distribution, the benefit is that it can be a fast hardware-based implementation.

The proposed approach is considerably more scalable than using a single front-end node that performs content-aware request distribution based on LARD or other algorithm. The distributor's job, the establishment and handing off of TCP connections, is typically

the bottleneck in this kind of switches, and making it distributed better scalability can be achieved paying only a small latency penalty. The centralised dispatcher could be a potential bottleneck as well, but it is not in practice as it is a very simple component that can be implemented in a dedicated node and handle up to 50,000 connections per second [4].

2.3.5 WARD

Workload-Aware Request Distribution (WARD) is a content-aware request distribution strategy that intends to improve some shortcomings of the scalable LARD architecture. The basic idea behind WARD is that "by mapping a small set of most frequent files to be served by multiple number of nodes, we can improve both locality of accesses and the cluster performance significantly" [39].

In a cluster using LARD of n nodes, only 1/n of the requests will be handled locally forwarding the other (n-1)/n requests, a relatively expensive operation [4] that can introduce a forwarding overhead that limits the performance and scalability of the cluster. Additionally, Web server workload studies have shown that 90% of the web requests are for only 2-10% of all the accessed documents, constituting less than 5% of the total working set. WARD aims to minimise forwarding overhead from handling the most frequent files while keeping the benefits of the optimised use of the overall cluster main memory caches that LARD provides.

WARD chooses a set of most frequently accessed documents, called core, that will be served by all the cluster nodes and partitions the rest of the files to be served by the backend servers.It uses an algorithm called *ward-analysis [39]* that identifies the optimal core for a given workload and system parameters such as the number of nodes, main memory cache size, TCP hand-off overheard and disk access overhead. The core is proposed to be computed once in a day, so that the cluster will always use the values obtained from previous day's workload.

This basic form of WARD is very similar to the scalable LARD architecture, and it would use the LARD distribution algorithm for the files that are not in the core. The other change proposed in WARD is to decentralise the dispatcher, by using a precomputed partitioning of the working set based on the workload of the previous days. The scalable LARD architecture introduces the idea of a co-located distributor that resides in every backend server, but still keeps a centralised dispatcher that every node has to contact to decide which backend server will handle the request. With a decentralised dispatcher, the overhead of a unique dispatcher is eliminated and better scalability is achieved. This approach will work well assuming that the core is reasonably stable, if workload stability is not possible, the simpler "dynamic" centralised dispatcher will be a better solution most of the time.

Simulation tests [39] show that WARD provides performance improvements of up to 260% increased throughput compared to round-robin strategies and up to 50% compared to locality-based strategies such as LARD. The use of a distributed dispatcher together with previous day's workload information improves throughput furthermore for typical stable workloads.

2.3.6 PPBL

Most web cluster architectures use a push approach to request distribution, the dispatcher decides which backend server should handle a given request based on layer 3, 4 or 7 information and then sends or "pushes" the request to the chosen server. WARD [39] introduces a distributed dispatcher, but it does so at the expense of having a static dispatching information that is updated with previous day's information instead of more dynamic algorithms like LARD [17]. The PPBL prototype web proxy uses a Distributed Shared Memory (DSM) based cluster interconnected by a Scalable Coherent Interface (SCI) network. It uses certain features of the DSM architecture to obtain better performance, and it is uncertain if this design would be able to be generalised to be appropriate for non-DSM clusters.

Parallel Pull-Based LRU [26] is an architecture that aims to distribute the dispatcher while still having the advantages of a dynamic content-based request distribution algorithm. As its name implies, it uses a "pull" approach to request distribution, where backend server nodes don't just receive requests but take part in a parallelised request distribution algorithm. This algorithm uses the basic idea that the backend servers, pull the requests that they should handle from an incoming request queue that resides in the front-end node.

The algorithm works as follows: each backend node selects a pending URL from the incoming queue and checks if it is in its own URL list. If the matching entry is found, the backend modifies the incoming queue to signal that it is handling the request and sends the reply directly to the client. The front-end can delete the entry from the incoming queue once a backend node has decided to handle it. If the entry is not found by any of the backend servers, the front-end will choose the less loaded node to handle it. The load information is kept in the DSM and periodically updated by each backend. If the backend that has found the request in its local URL list is overloaded, it can choose a less loaded in the entry's server list or if there is none, it can forward it directly to the less loaded node in the cluster keeping all the information about the URL local, accessible to the other node through the DSM.

A first implementation that used the above design proved not to be very scalable. Throughput doesn't improve anywhere near linearly by adding nodes to the cluster, and the synchronisation locks required by the request distribution algorithm prevent the cluster

from performing better. A second implementation is proposed by the authors that will limit the number of locks that are necessary for each request. This implementation uses new data structures and an special SCI mapping that performs a fast atomic fetch and increment on each access that can be used by the backend nodes to signal when they have finished evaluating if a given request is for them. Good scalability is achieved by making use of these low level capabilities of the middleware, a cluster of 7 nodes has a 580% better throughput than a single node server.

PPBL is a very interesting design that takes a new approach to request distribution, both because it uses a pull based approach whereas most other cluster architectures use a push design as well as because it makes use of high-speed SCI networks and DSM middleware to achieve better scalability. It would be interesting to evaluate if the ideas behind PPBL are applicable outside of the specific implementation technologies used by the authors, as they had to resort to low-level optimisations to achieve reasonable scalability.

2.4 Web caching replacement policies

Web caching used with HTTP proxy servers is a widely used and well studied [3, 8, 7, 27, 28] mechanism for reducing both latency and network traffic in accessing resources on the Web. In the context of a Web server, the purpose of a cache and its associated replacement policy is to maximise the number of requests that are handled with data available in the main memory of the server, avoiding thus costly file-system input-output operations.

Modern computer main memory and high speed networks feature access speeds which are still considerably faster than the speeds of magnetic hard disks, making the management of an in-memory cache an important and frequently overlooked part of the design of a clustered web server. Web server cluster designers, both research-oriented and commercial, acknowledge the benefits of taking in account locality of references in Web traffic for added performance, but the majority of proposed solutions deal with this issue only at the request distribution level. Unlike with Web proxies, there is little previous research on cooperative caching strategies for Web clusters.

The main aspect that determines the performance of a cache is its cache replacement policy. This algorithm will be in charge of deciding which document(s) is/are evicted from the cache when its capacity has reached its limits and a new document has to be loaded and maybe cached by the server. Before the Web, cache replacement policies have been studied for file-system caching and virtual memory page replacement, but Web caching has different requirements than those domains, most importantly that Web objects (HTML pages, image files, etc.) vary widely in size. Furthermore, Web cluster caches differs from Web proxy caching in that network traffic use is not a big concern for clusters,

the nodes are normally interconnected using high-speed private networks, so for a given node it is always preferable to obtain a resource from the main memory of another node instead of reading it from the file-system.

When evaluating Web proxy cache replacement policies several performance metrics are commonly considered:

Hit rate The fraction of the total requests that are served from the local cache.

Byte hit rate The percentage of network traffic that were sent directly from the cache, without involving file-system operations.

Optimising one of these two metrics doesn't optimise the other, so a trade-off has to me made, depending if CPU cycles or network bandwidth is the bottleneck. Byte hit rate is important for Web proxies that need to minimise the traffic between the proxy and the original servers. In a common deployment scenario of a Web proxy such as in a University campus, there are important costs in latency and/or money for each byte part of a request that requires a connection to the original server, while Web clusters do not have this concern, as the nodes in a cluster use high-speed interconnects. Thus, the most important metric when evaluating cache replacement policies for Web clusters is hit rate.

2.4.1 Least Recently Used (LRU)

Least Recently is a very popular policy in file system and virtual memory caching, it evicts the document which has been least recently accessed from the cache. In the absence of cost and size concerns, LRU is the optimal on-line algorithm for requests sets with good locality. However in Web caches, replacing a recently used big file can give better hit ratio than replacing a less recently used but small file. Thus, for a good Web cache replacement policy size has to be taken into consideration and the basic LRU is not a good option because it ignores the sizes of cached documents.

There are several variations of LRU that try to improve its performance:

LRU Size

[3] It is a variant of LRU [3] that tries to minimize the number of documents replaced by taking the size of them in account as well as the last access time. It will always remove the larger documents first, and fall back to LRU in the case of having several documents of the same size.

Log(Size)+LRU [3]

This variation of LRU-Size uses a logarithm of the size, to avoid giving to much weight to the size aspect of a document, so that the LRU order is used more frequently than in

the case of LRU-Size.

LRU-Min

This policy [3] gets a list of documents that are of bigger size than the requested document, and evict them in LRU order until there is enough free space on the cache to fit the new request. When there are no files bigger than the requested document left in the cache, LRU is used to choose the next eviction candidate(s).

LRU-Threshold

Another variant of LRU that will avoid caching documents that are too big by setting a threshold size so that no document larger than that size will be cached [3].

Segmented LRU

Segmented LRU is a frequency-based variation of LRU designed for fixed-size page caching in file-systems. Observing that objects with two accesses are much more popular than those with only one access, the cache space is partitioned into two LRU segments: probationary segment and protected segment.

Objects brought to the cache are initially put in the probationary segment, and will only be moved to the protected segment if they get at least one more access.When an object has to be evicted, it will be taken from the probationary segment first. The protected segment has a fixed size, and when it gets full the objects that don't have space in it will be kept in the probationary segment.

Segmented LRU is not suitable for Web caching as it ignores the size of cached objects and assumes fixed-size objects. Furthermore, it has the problem of needing parametrisation of the number and sizes of segments.

2.4.2 Least Frequently Used (LFU)

This policy keeps track of the number of requests that are made for each document in the cache, evicting the document of documents that have been less frequently requested first if space is needed. As it ignores the document sizes, it can lead to an inefficient use of the space on the cache. Where LRU is equivalent to sorting by last access time, LFU is equivalent to sorting by number of accesses.

2.4.3 Hyper-G

It is a policy that combines LFU, LRU and the size aspect [28]. It uses a hierarchical approach, where LFU is always applied first, if a tie happens LRU is used and finally if

there is still more than one candidate the size of the documents is taken in account for deciding which one to evict.

2.4.4 Pitkow-Recker

It uses a different primary key depending whether or not all cached documents have been accessed in the current day [28]. It will use the LRU policy, except if all the documents have been accessed today, in which case it uses a Size removal policy, removing the largest one(s).

2.4.5 Hybrid

The Hybrid policy [27] has been designed to minimise the time that end-users wait for a page to load as well as hit rate and byte hit rate. It is a hybrid of several factors, considering download time, number of request (frequency) and size. Hybrid selects for eviction the document *i* with the lowest value for the following expression:

$(clat_{ser(i)} + W_B/cbw_{ser(i)})(nref_i{**}W_N)/S_i$

where $nref_i$ is the number of references to document i since it last entered the cache, s_i is the size in bytes of document i, and W_B and W_N are constants that set the relative importance of the variables $cbw_{ser(i)}$ and $nref_i$, respectively. A document won't be evicted if the expression above evaluates to a large value, which could occur if the document is in a server that takes a long time to connect and is connected via a low bandwidth link, if the document has been referenced many times in the past and if the document size is small.

Although this replacement policy has the advantage of considering the latency of a document, this is only applicable to Web proxy caching. Retrieval latency is not a concern on Web server clusters, as all the nodes are expected to be interconnected by the same kind of network and the latency should be the same for all of them.

2.4.6 Lowest Relative Value (LRV)

LRV is based on the relative value (V), a function of the probability that a document is accessed again (P_r). The LRV algorithm simply selects the document with the Lowest Relative Value as the candidate for eviction. As V is proportional to P_r, the issue is to find this probability.

The parameters used for computing P_r are the following:

- Time from the last access.
- Number of previous accesses.
- Document size.

In the simulations carried out by Rizzo and Vicisano [19] LRV features higher byte hit rate than other policies in all conditions, and has better hit rate than all of them except SIZE.

2.4.7 GreedyDual-Size

GreedyDual-Size is a variation of GreedyDual, an algorithm for uniform-size variable-cost cache replacement [12]. GreedyDual deals with the case in which cached pages are of the same size but have different associated costs for bringing them into the cache. It associates a H value with each cached page, initially set to the cost of adding a page from secondary storage to the cache. When a replacement has to be made, the page with the lowest H is evicted, and all the rest pages reduce their H value by the evicted page's H value.

GreedyDual-Size incorporates the size factor by setting the initial H value of a document to *cost/size*, where *cost* is the cost of bringing the document to the cache and *size* is the size of the document in bytes. There are different versions of GreedyDual-Size that differ on the definition of *cost*, depending if the goal of the replacement algorithm is to maximise hit rate or byte hit rate. GreedyDual-Size(1) sets the *cost* to 1 and achieves the best hit rate while GreedyDual-Size(packets), which sets the *cost* to *2+size/536* (the estimated number of network packets sent and received if a cache miss happens) maximises byte hit ratio. GreedyDual-Size(1) has very good hit rate at the price of lower byte hit rate, while GreedyDual-Size(packet) achieves the highest byte hit rate with moderately lower hit rate than GreedyDual-Size(1).

Although GreedyDual-Size(packets) would be the recommended policy for the overall best performance in Web proxy caches, in the case of a Web cluster GreedyDual-Size(1) would be more appropriate as it achieves better hit rate by not paying attention to the byte hit rate. This replacement policy can be considered the current "champion" of web cache replacement algorithms.

Chapter 3

Design and implementation

Web clusters have been traditionally implemented by "growing" single-node Web server architectures instead of being designed from scratch as a system that takes advantages of a clustering architecture, clustering has been more "bolt-on" Web servers than "built-in". This legacy reflects on the design of commercial Web clustering and load balancing solutions. A very common deployment scenario for a commercial clustered Web server configuration is shown in Figure 3.1, where the dispatcher is the central point of contact for all the clients and it distributes requests to the backend Web servers using layer 4 switching with either layer 2 or 3 forwarding. There are some advantages to this design:

- Backend nodes are standard Web servers, keeping the same configuration and administration needs of a single-node Web server, minimising the system administration overhead of moving from a single Web server to a cluster.
- The front-end dispatcher can be a dedicated device with embedded software that is very efficient at the request distribution and load balancing tasks it performs.
- Every component in the cluster is transparent of the others, it is a loosely-coupled architecture where each component has no or minimal knowledge of how the others work, providing both fault tolerance and simpler administration and deployment, as well as allowing heterogeneous systems to work together.

The architecture in Figure 3.1 is simple and provides some degree of scalability. However, goals of higher scalability and better use of resources available on the backend nodes require different solutions. Most of the commercial state-of-the-art Web switch/dispatcher products have the optional functionality of taking in account application layer (OSI layer 7) information, such as the requested URL or the current session.

Application layer data awareness (layer 7 switching) is important for achieving better request distribution , but commercial dispatchers pay a big performance price when doing it as they have to be both transparent to the clients as well as to the backend Web servers.

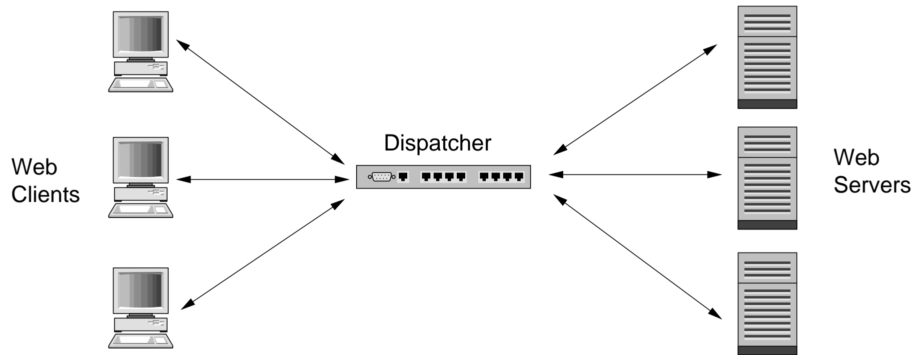


Figure 3.1: Common Web cluster deployment scenario.

The fact that these clustering solutions are highly transparent between their own components (dispatchers and backend nodes) as well as to the outside world (Web browsers) makes it difficult to implement highly scalable clustered Web servers with them.

In this section the design and implementation of a clustered Web server that uses commodity off-the-shelf (COTS) hardware and distributed software components is described. Designing all the components from scratch allows more flexibility for using state-of-the-art algorithms and controlling resources globally in the cluster, achieving better load-balancing and efficiency, which should lead to good scalability. This Web cluster could be considered as a Web server with an integrated content-aware layer 7 switch and global cache management.

All the clustering software discussed in this dissertation has been implemented in Java. It is not very common in the research literature to use Java as the platform for implementing prototype Web servers or proxies. However, there are popular commercial application server products implemented in Java that prove that it is a viable platform for high-performance network servers. Furthermore, with the release of Java Standard Edition version 1.4, non-blocking I/O support is made available to Java developers both under Unix as well as Win32 operating systems.

This chapter is organised as follows: first the overall architecture of the clustered Web server is described; next each of the components' (front-end/dispatcher, node manager and cache manager) design is discussed, giving details of their different implementation approaches and how they interact with the rest of the cluster components.

3.1 Architecture

Three main components form the clustered Web server discussed in this dissertation:

Front-end The front-end is the process that will be the single point of contact for all the clients that want to issue an HTTP request to the server and receive a response from it. Every byte of data that the server receives or sends has to go through it,

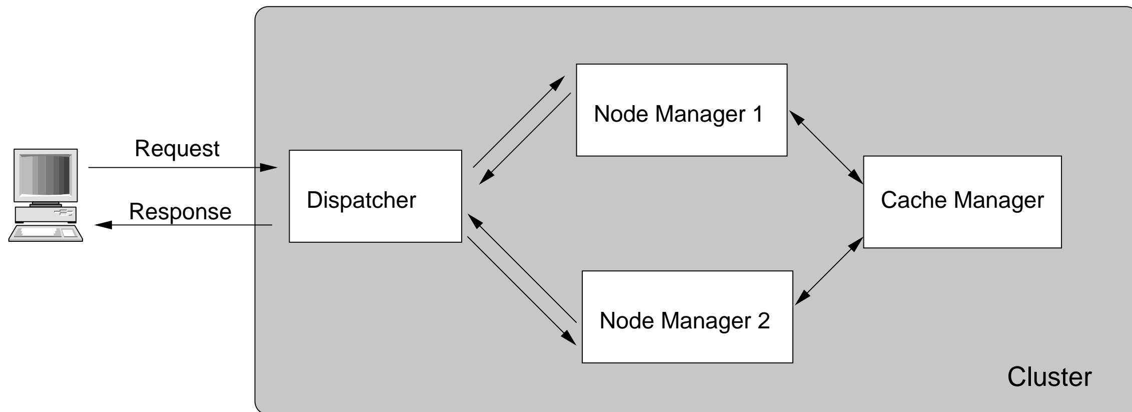


Figure 3.2: Logical view of the components in a running cluster.

so it is the most performance-sensitive component of the server. It has three main responsibilities:

1. Wait for connections on a TCP port and accept requests from Web clients, keeping track of the open client connections.
2. Obtain the request URLs and decide which backend node will handle them.
3. Receive and send back the responses generated by the backend nodes to the appropriate clients.

Cache manager The cache manager is the component that holds cluster-wide global information about the state of the backend nodes' main memory cache. Its main responsibility is to keep a mapping of files and the nodes that have them them in their cache, so that when a backend node has a request for a file that is not in its local cache it can retrieve it directly from another node's main memory instead of accessing the file-system. The reason for this design is that with the current network interconnection technologies it should be faster for a node to retrieve some data from another node's main memory across the network than reading it from the file-system.

Node manager The node manager is the main "worker" process in the cluster. Each backend node runs one or more node manager process. Its main responsibilities are:

1. Retrieve requests assigned to it by the front-end, obtain the response data and send it back to the front-end.
2. Keep a local main memory cache of files, with its associated cache replacement policy.

There are two possible views that can be used when describing how these components (or instances of them) are organised in a running clustered Web server to handle client requests. Figure 3.2 shows a logical view where one front-end, two node managers and one cache manager work to accept a request, route it through the cluster software and send the right response back to the client.

Figure 3.3 describes the same cluster configuration as Figure 3.2 but from a physical point view, showing how the component processes could be distributed in different computers. In this case there is one system dedicated as front-end, another one running the cache manager process and two nodes running one node manager process each. This configuration is the one that will be evaluated in this dissertation, with the only difference of the number of systems running node manager processes.

The server software is flexible with respect to the physical distribution of its components. It requires the existence of one front-end, one cache manager and one or more node manager, but does not require them to be physically distributed among the nodes of the cluster in a certain way. It is possible for example to have all the components running in the same system, even though that would obviously not be recommended.

Another aspect in which the cluster software offers high flexibility is that different implementations of its subsystems can be used by specifying a command line option at startup time. For example, there are multi-threaded and non-blocking I/O implementations of the front-end server, and there are different request distribution and cache replacement policies available as well. It is a framework that can easily integrate new im-

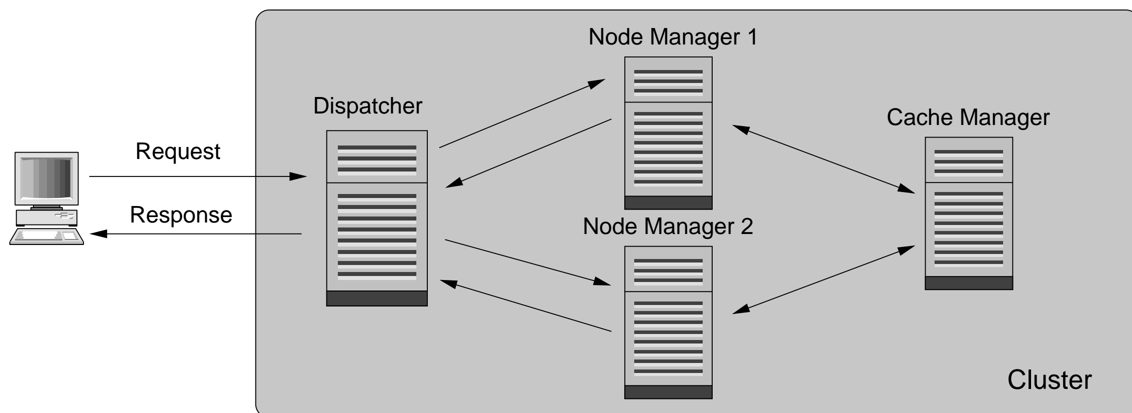


Figure 3.3: Physical view of the components in a running cluster.

plementations for most of its parts, with clearly identified interfaces between the different subsystems so that they can inter-operate.

3.2 Inter-process communication

The Web cluster software described in this dissertation is designed to run in a cluster of computers connected by a network that supports TCP/IP. Thus, there is a need to define a set of protocols that the different components - front-end, cache manager and node managers - will use for communicating between them. For example, each node manager needs to have a protocol that allows it to contact the front-end and retrieve HTTP requests that the front-end has assigned to it. In the same way, there has to be an agreed-upon protocol between node managers and the cache manager so that they can collaborate to implement an effective caching policy that takes advantage of the aggregated main memory cache available in the cluster.

Two alternative inter-process communication protocols have been implemented in the cluster, the first one uses Java interfaces via Java Remote Method Invocation (RMI) and the other one uses a custom application-specific protocol based on low-level TCP socket connections. These two protocols will be described in the following sections.

3.2.1 RMI protocol

Remote Method Invocation (RMI) is a Java-specific middleware for implementing distributed object systems. It allows Java programs residing in different address spaces and potentially different machines to invoke object method calls between themselves. RMI makes it possible to ignore up to a certain extent that the object that will receive a given method call can possibly be in a different machine across the network.

The main advantages of RMI are that it is a standard part of the Java platform and that it makes it very simple to specify inter-process communication protocols using Java interfaces. RMI servers are simply objects that implement a certain remote interface and "export" it to the RMI subsystem, making those objects accessible by their interfaces to remote Java processes.

However, as will be later discussed, RMI has shown to not be suitable for implementing high-performance scalable servers. It uses a thread and an associated TCP connection for each method call, an approach with very poor scalability. Threads and TCP connections are expensive resources with high overhead, so the standard RMI implementation is not suitable for systems that are expected to handle hundreds or even thousands of method calls per second. Even if Java has introduced new much more scalable non-blocking I/O in 1.4 the rest of the APIs that come with the Java Development Kit (JDK), RMI included, do not use it. A reimplementation of RMI using non-blocking I/O and multiplexed TCP connections would possibly be more scalable, BEA WebLogic uses a proprietary implementation of RMI that does this.

3.2.2 Socket-based application-specific protocol

The use of an application-specific inter-process communication protocol provides two main advantages compared to RMI for the development of the Web cluster:

- The protocol can be much more efficient, only transmitting the strictly necessary data between the processes and avoiding the generic approach of RMI and its associated serialization and deserialization overhead. RMI implements a subsystem that has to support any kind of legal Java method call and parameters, using a generic object serialization format to transfer information from one machine to the other. The application-specific protocol can be more efficient by only supporting specific kinds of messages between the processes.
- Use of the new Java non-blocking I/O facilities and event-driven architectures. The current implementation of RMI uses blocking TCP sockets and a thread-per-connection model, which limits its scalability. When implementing an application specific protocol based on raw TCP sockets, it is possible to take advantage of the new Java I/O and to implement it using an event-driven approach that provides better scalability than using one dedicated thread per message.

The main reason for using the socket-based protocol is to achieve a good scalability and throughput in the cluster, at least better than what is possible with an RMI-based approach. For this reason, where RMI would cause the establishment of several TCP connections between the front-end and the node managers and the node managers and the cache manager, in this design each cluster component has only one open TCP connection to any of the other components it has to communicate with. The front-end will have a connection to each one of the node managers, and the node managers will all have one connection to the cache manager as well.

These connections will have associated non-blocking SocketChannels that will deliver I/O events to the main loop of the components. The use of non-blocking I/O imposes significant changes in the design of the concurrency model used on the different components, reducing the use of threads to the strictly minimum. This aspect of the interprocess-communication in the cluster will be discussed in the next section.

We will not detail the structure of every the possible message sent between the components, but the format of the messages sent from the dispatcher to the node managers will be described as an example of the protocol design approach used. Figure 3.4 shows the format of the messages that the front-end sends to the node managers when it is handing-off client requests for the node managers to fulfill. The message has a fixed-size header part and variable-size message data or payload part. The header is composed of eight bytes, which contain two 32 bit integers, the first one specifies the length in bytes of the

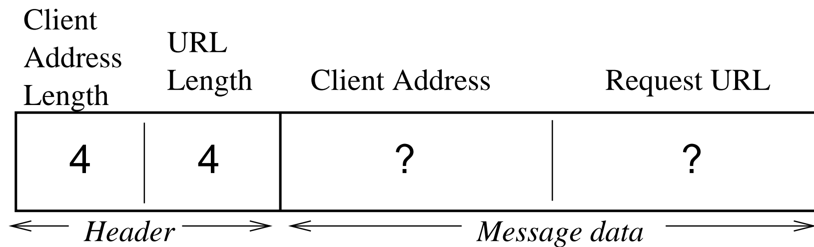


Figure 3.4: Structure of the messages sent from the front-end to the node managers.

client address field that comes after the header and the second one gives the number of bytes that the requested URL takes after the client address. The client address in the data part contains a text string that uniquely identifies which client made a certain request, this is necessary so that the front-end knows which client should receive a response. The second field of the data part contains the request URL, which will eventually be mapped to a file in the file-system by the node manager.

The structure of the message described above shows how the protocol between the dispatcher and the node manager allows for information of variable size to be sent to the node manager, but does it in an efficient way, without all the space and processing overhead of serialising and deserialising Java String objects.

3.3 Concurrency model

In the Web cluster design described in this dissertation all the components have to be able to handle as many simultaneous requests as possible: the front-end receives requests from the Web clients, node managers receive requests assigned to them by the dispatcher and the cache manager receives queries and updates from the node managers. It is extremely important that each one of the component is able to handle several simultaneous requests, and does so in an as efficient as possible way.

There are two main approaches to implementing the concurrency or process model of the components, a thread-per-connection model that was most common in Java servers until recently and an event-driven approach which has only been made possible with Java 1.4 and the introduction of the new non-blocking I/O facilities.

3.3.1 Thread-per-connection model

The thread-per-connection model is typically used in Java network servers to be able to handle simultaneous client requests. The basic idea behind this model is that each service request is handled by its own dedicated thread. When used for Web servers, this means that as soon as the server accepts a TCP connection from a Web client, it will assign the

servicing of any request that comes through that connection to a “handler” thread, so that the main server loop can keep accepting new connections.

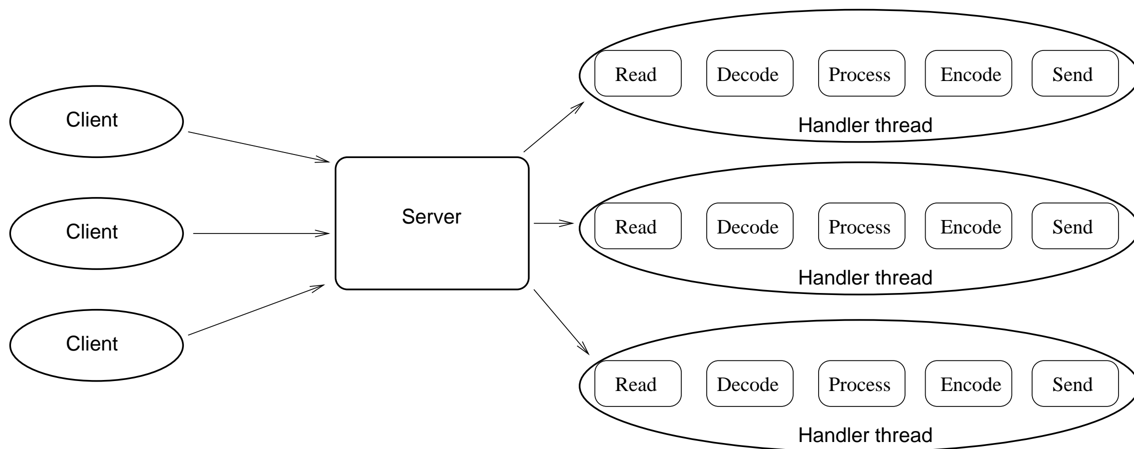


Figure 3.5: Thread-per-connection model.

Although it is a straightforward and common design, the thread-per-connection model has its own problems. The throughput and scalability of the server will suffer due to threading overhead. Each thread has its own stack and receives some CPU allocation, and thread switching can be considerably expensive if hundreds or even thousands simultaneous connections are expected to be handled. Furthermore, threads would only represent a performance and scalability improvement when used in systems with more than one CPU, in single-CPU machines threading can help in handling more than one task or request at a time, but paying always a cost in throughput and scalability. Ideally multiple threads should only be used in multi-processor machines, and using up to one thread per processor, but they are frequently used for overcoming the limitations caused by the Java’s until recently only blocking I/O facilities.

3.3.2 Event-driven model with multiplexed I/O

Until Java 1.4, the standard I/O facilities offered by the Java platform where all blocking, so the only way to handle simultaneous requests clients was to use a multi-threaded design, with its scalability limitations. The introduction of non-blocking I/O in Java 1.4 has changed this, allowing the implementation of event-driven designs like the one described in this section.

For single-processor Web servers and Web clusters built with single-processor systems, use of multiple threads does not provide good scalability and should be avoided if possible. Threads, when used for monitoring multiple sockets and blocking on accept/read/write operations on them, are just performing the job of a socket monitor and the thread scheduler is working as the notification mechanism. The problem here is that

they were not designed for this use, and there are big performance costs for the Java virtual machine in managing multiple threads.

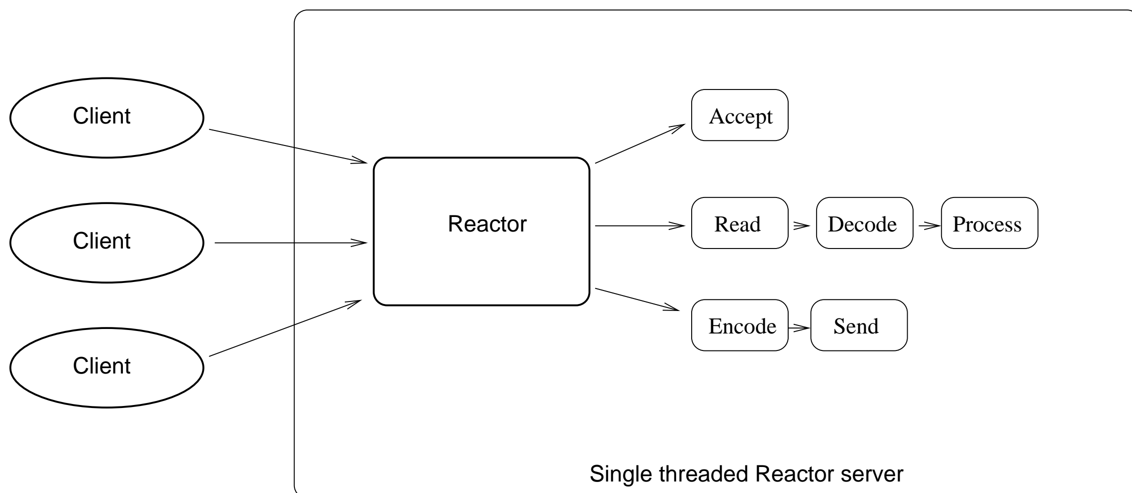


Figure 3.6: Single threaded event-driven model with multiplexed I/O.

The new I/O APIs available in Java 1.4 provide facilities for non-blocking I/O and a built-in mechanism called Selector that implements the Reactor pattern explained in the next section.Using these two mechanisms, servers can be notified when I/O operations are available, with no need for the servers to dedicate threads to blocking on sockets or files. This design is called “event-driven” because the server application does not initiate and block until I/O operation, it only registers its interest in one or more operations and gets notified when they can be performed in a non-blocking way. It is a design similar to how GUI event-loops work.

Server applications that use a event-driven multiplexed I/O model are designed quite differently from thread-per-connection servers, as seen in Figure 3.6. In thread-per-connection servers the “unit of work” is the whole client connection, a single thread handles a connection from when it is accepted until it is closed. In event-driven servers, the server handles several different I/O “events”, performing each time a small non-blocking operation that handles the received event. For example, if a Web server receives a WRITE event on a SocketChannel, it means that it can at least write something to that client socket without blocking. When a non-blocking write method is called on the socket channel, and depending on if all the bytes of the response have already been written or not it could close the channel if those are the semantics of the protocol between the server and the client.

Reactor architectural pattern

The Reactor pattern allows event-driven applications to demultiplex and dispatch service requests that are received by an application from one or more clients. It inverts the flow of control typical in traditional servers, in the sense that a component called a reactor

is responsible for waiting for external event notifications and dispatching them to the appropriate event handlers, the application does not actively wait for events, it is instead notified when they happen.

Reactor defines the concept of event handlers that process certain types of events from certain event sources. Even handlers are registered with a reactor, specifying the concrete event types and sources they are interested in. When a given event occurs, the reactor will notify the event handler registered for that concrete event type and source of its occurrence, so that it can handle it in an application-specific way.

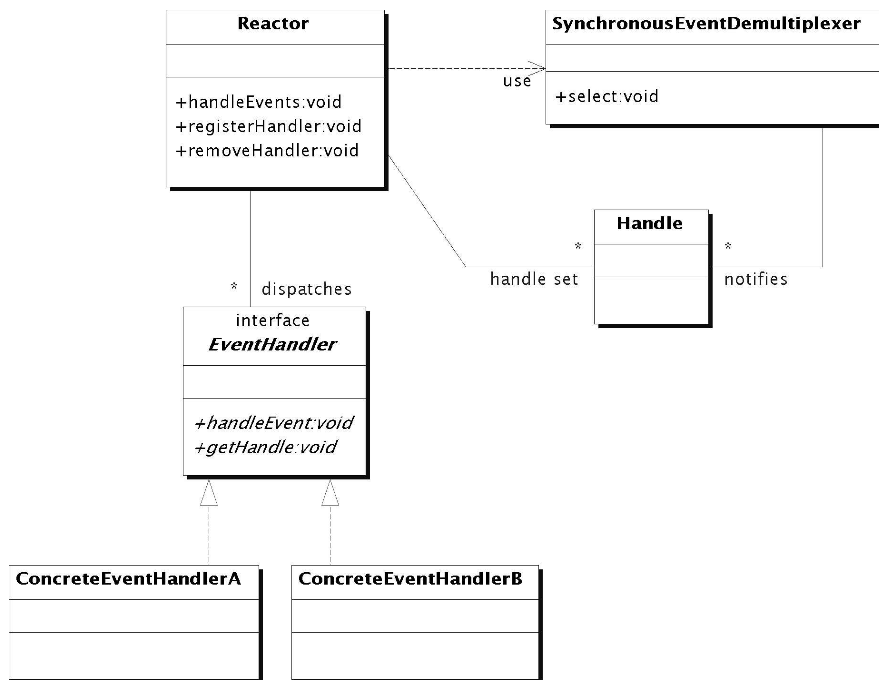


Figure 3.7: Class diagram of main participants in the Reactor pattern.

There are five main participants in the Reactor pattern (see Figure 3.7):

Handle Identifies event sources such as network connections or open files that can generate events. Events can be for example ACCEPT and READ for handles associated with network sockets or WRITE for handles that are linked to file I/O. In Java, handles are represented by the java.nio.Channel interface and all the classes that implement it.

Synchronous event multiplexer It is the function that waits for one or more events to occur on a set of handles. It will block until there is at least one handle in its set that is ready, meaning that it can perform I/O operations without blocking. An example of a synchronous event multiplexer is the select() function support by many POSIX-like operating systems.

Event handler EventHandler is an interface that defines a set of hooks or callback methods that should be called when a certain indication event occurs.

Concrete event handler Concrete implementations of EventHandler implement application-specific logic to process the indication events received through their associated handle. Each concrete event handler is associated with a certain handler that uses as an indication event source.

Reactor The Reactor is the "registry" that keeps track of event handlers and their associated handles. When an event occurs in the handle set of the Reactor, it finds the associated handler that is interested in the occurred indication event and it calls the appropriate method of the concrete event handler so that it can run whatever application-specific code that should be run for the event. In the Java APIs, the *java.nio.channels.Selector* class performs the function of the Reactor.

3.4 Front-end

The front-end is in charge of accepting connections from the clients, distributing them between the different backend nodes and sending the responses back to the clients. The front-end deals with every byte received and sent by the cluster as well as with all the client connections. It can be said that the front-end is the component of a Web cluster that determines its scalability. There are important aspects that can be handled by the backend nodes such as global main memory cache management and request pulling approaches, but the front-end is more likely to be the bottleneck most of the time.

In the design implemented in this dissertation, the front-end performs the minimum tasks that any centralised layer 7 HTTP switch has to perform:

1. Accept connections from web clients.
2. Read the HTTP request from the clients and determine which is the request URL.
3. Decide which backend node will handle each of the requests.
4. Send the responses back to the clients when the backend nodes have finished processing them.

The front-end has been implemented as a Java daemon that listens on a certain TCP port. The daemon has two main subsystems with different pluggable implementations of them:

Socket listener This is the part of the front-end that deals with accepting client connections, reading the data they send and passing it to the dispatcher. It defines the concurrency model that the will be followed by the front-end, that is, how many

concurrent threads will run and which are their responsibilities. The socket listeners' interface is specified by the *ISocketListener* Java interface, and two alternative implementations of it are discussed in this dissertation: a multi-threaded socket listener and a socket listener based on the Reactor architectural pattern that uses non-blocking I/O.

Dispatcher The dispatcher's main responsibility is to decide which one of the node managers should handle a given request. There are several different implementations of the *IDispatcher* interface available in the cluster. A secondary function of the dispatcher is to serve as a point of contact for the back-ends' node managers when using the RMI-based communication protocol.

3.4.1 Multi-threaded socket listener

The multi-threaded socket listener is one of the two implementations of *ISocketListener* that will be discussed in this dissertation. This implementation uses the thread-per-connection model. However, this socket listener does not just implement an infinite loop that spawns a new thread for each accepted connection, as that would be too inefficient. Instead, a thread pool is used so that thread instances are recycled as much as possible.

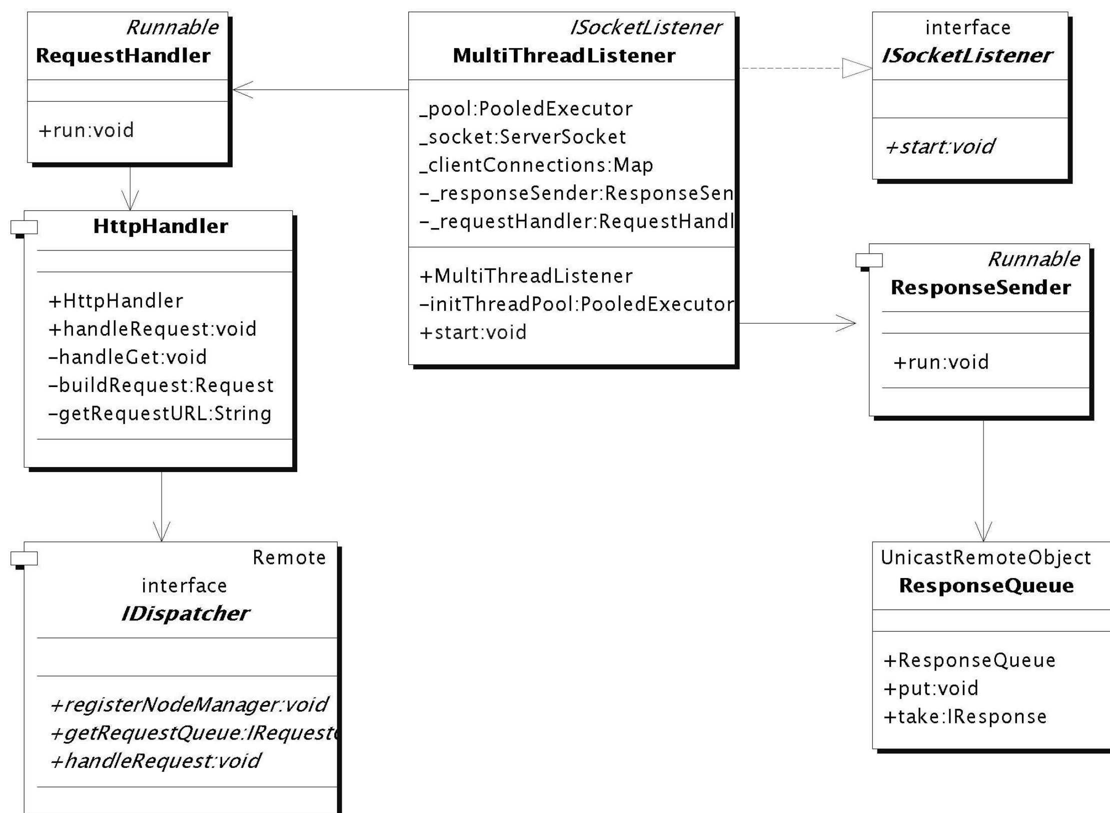


Figure 3.8: Multi-threaded listener class diagram.

Figure 3.8 shows the main actors in the design of the multi-threaded listener. The *MultiThreadListener* class itself does little more than setting up the thread pool and starting two main threads that will do most of the work, a *RequestHandler* and a *ResponseSender.*

The *RequestHandler* thread runs an infinite loop that accepts connections from clients and handles each connection by sending it as a *Runnable* which is a unit of work that the threads pooled in the instance of class *PooledExecutor* carry out, as seen in Figure 3.9. When the pool receives the task, it will assign a thread to it and call the *handleConnection* method of the *RequestHandler*, which in turn reads the raw bytes of the request sent by the connecting client and passes the data on to the *HttpHandler* instance.

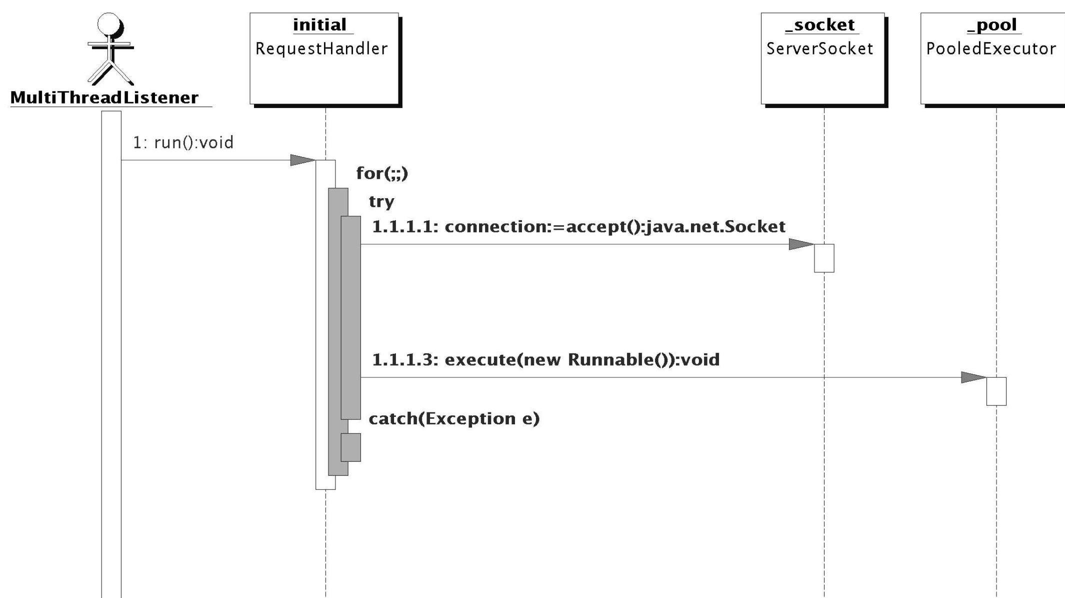


Figure 3.9: Sequence diagram for RequestHandler thread.

The *HttpHandler* is the object that holds all the knowledge necessary for understanding the clients' HTTP requests, it parses the request data to verify that it is correct and obtains the URL that the client is interested in. Once the URL is known, the *HttpHandler* simply delegates the handling to its associated *IDispatcher* implementation, that will decide which one of the registered *NodeManagers* should handle the request and put it in its request queue.

The multi-threaded socket listener has some mechanisms for avoiding performance penalties associated to the thread-per-connection model:

- Minimisation of thread creation overhead, by using a *PoolExecutor* that processes work items (eg. HTTP requests) backed by a pool of threads. This can be a less important factor if threads are pooled internally by the JVM, and it could even be counter-productive in that case.

- Use of *RequestQueues* for handling requests to backend nodes. This avoids the front-end threads to block until a backend node is available for collecting a new requests, it decouples the decision of which backend node should handle a given request (producer) from the handling of the request (consumer). This is a common design idiom for event-based concurrent systems [6].

3.4.2 Reactor-based socket listener

In this section an implementation of *ISocketListener*, the performance-critical "engine" behind the front-end, that uses non-blocking I/O is described. As this implementation is based on the Reactor architectural pattern.

The ISocketListener implementation that uses the new Java non-blocking I/O APIs has a design approach that is quite different to the thread-per-connection model. The multi-threaded implementation divides the concurrent tasks by connection, associating a dedicated thread with each connection so that the clients can be served concurrently. The underlying idea here is that serving a request associated to a TCP connection is an indivisible work unit that is potentially long-running and its processing should be concurrent to all the other client connections. However, this is not always a good approach as in reality it is typical for network servers to be more I/O bound than CPU bound.

The implementation described here uses a minimal amount of threads to evaluate the performance of non-blocking I/O on its own as much as possible, but there is an endless possibility of variations that go from a single-threaded server to the use of a pool of worker threads for handling CPU-intensive requests on servers that run on multiprocessor machines.

The *ReactorSocketListener* uses only three threads for processing client requests, as seen in Figure 3.10:

Acceptor The Acceptor thread is registered to the ACCEPT events of the *ServerSocketChannel* that listens for incoming TCP connections. Its only responsibility is to accept the client connections store a reference of each client connection and put them in a queue and notify the Reader thread to register itself for the READ events on them.

Reader The Reader thread runs an infinite loop alternating between two operations:

- Registering itself for the READ operation events with the *SocketChannels* that represent new client connections.
- Reading request data when it is notified that there is some available by blocking on a *select()* call. This call will only return in two circumstances, when there are client connections ready with data to read from them or when the

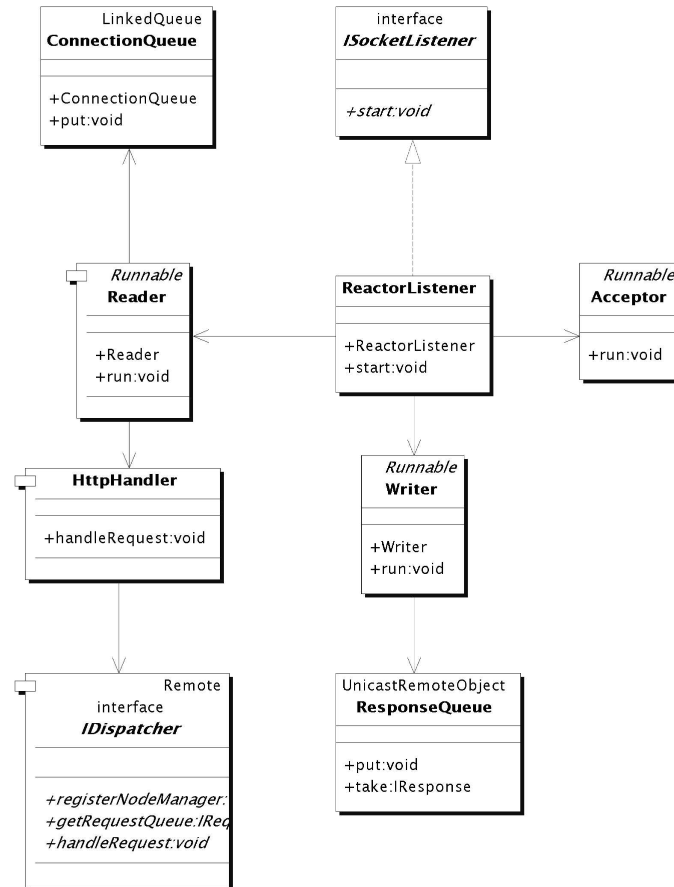


Figure 3.10: ReactorListener class diagram.

queue of new client connections has new connections for the *Reader* to register with.

The Reader thread itself only reads the request data as an array of bytes, it delegates further request processing to an instance of *HttpHandler*, so from the moment that it calls *HttpHandler::handleRequest()* the server follows exactly the same path as with the *SocketListener*, the request will go through the *HttpHandler* to an instance of *IDispatcher* implementation that will decide to put it in a *RequestQueue* associated to a certain *NodeManager*.

Writer The Writer runs an infinite loop with the task of taking responses that the NodeManagers put in the front-end's ResponseQueue and writing them to their associated client socket channels.

The *RequestQueue* is a remote RMI object that *NodeManagers* get a reference to when they register with the front-end. When they obtain the data that satisfies a given request, they create a *Response* object that contains this data as well as an identifier of the client connection that it should be written to.

The Writer has to simply loop forever taking *Responses* from the *ResponseQueue*, finding their associated client connection channel and writing to it the data that is in the *Response*'s responseData property.

3.4.3 Dispatcher

Where the SocketListener is the "engine" of the front-end, accepting and reading requests from client connections in an as efficient as possible way, the dispatcher implements the request distribution algorithm used for deciding which backend node will handle each request.

The *IDispatcher* Java interface has to be implemented by the any request distribution algorithm that could be used by the front-end, Figure 3.11 shows the class hierarchy for *IDispatcher*. There are different distribution algorithms implemented and other ones are easily pluggable by just implementing *IDispatcher* and chosen by the front-end by passing the desired implementation as a command line option at startup time.

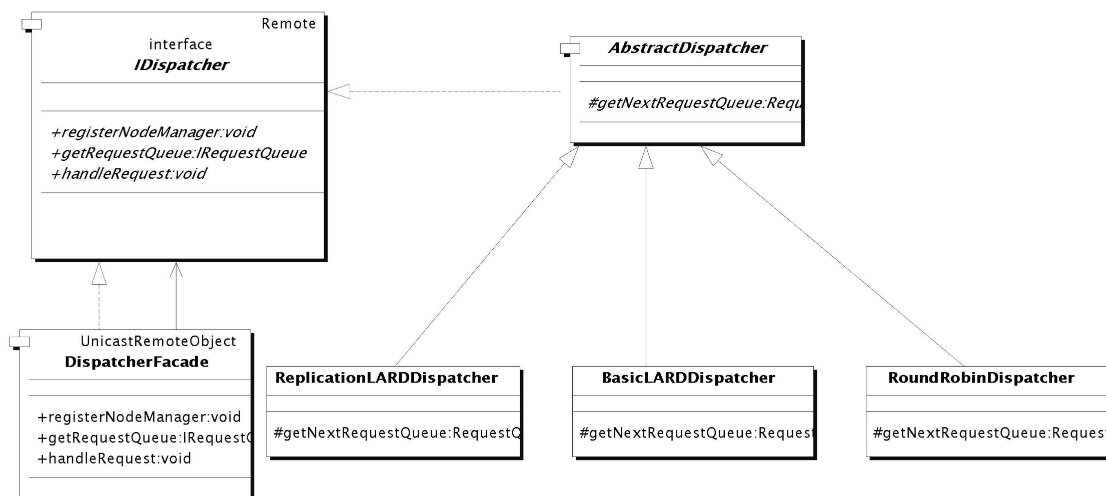


Figure 3.11: IDispatcher interface class hierarchy.

The concrete implementations of *IDispatcher* are:

DispatcherFacade This is an special implementation of *IDispatcher* that does not implement any request distribution algorithm by itself. It delegates any *IDispatcher* operation to another implementor and as it extends the *UnicastRemoteObject* class it is automatically a remote RMI server object.

At front-end startup time, the *HttpHandler* will always create an instance of *DispatcherFacade* as the main *IDispatcher* implementation. The DispatcherFacade in turn loads and instantiates another concrete *IDispatcher* implementor that is chosen by a command line option, delegating all dispatching decisions to it.

The reason for this slightly complex approach is that apart from the pure algorithm implementation aspect, the front-end needs to export a *IDispatcher* implementation that is remotely accessible for the backend nodes when the RMI-based inter-process communication is being used. This *IDispatcher* will be accessed by the back-ends for registering themselves with the front-end and obtaining the request and response queues.

It is not necessary for every *IDispatcher* implementation to be a remote RMI server, it is desirable to separate the concerns of request distribution algorithm and remote accessibility. The DispatcherFacade allows other concrete IDispatcher implementations to be implemented without any concern regarding the local or remote environment in which they will be accessed.

RoundRobinDispatcher The simplest of the IDispatcher implementations, it uses an standard Round-Robin algorithm. The reason for making available such a simple and clearly not optimal algorithm is to use it as a base benchmark comparison when evaluating other more sophisticated request distribution algorithms.

BasicLARDDispatcher The first of the LARD variants implemented follows the Basic LARD algorithm explained in section 1.3.4. It is a locality-aware request distribution algorithm that maintains a mapping of requests and the backend nodes that have handled them in the past. It has considerably more overhead than the round-robin version, but it should offer better performance and less cache misses on the back-ends.

ReplicationLARDDispatcher A variation of Basic LARD that implements a locality-aware distribution algorithm with replication. That is, it allows to have more than one backend node associated to a given request, effectively "replicating" certain pages in several backend node main caches. Although an improvement compared to Basic LARD, it is not clear if its added complexity to a performance-critical front-end is acceptable.

3.5 Cache Manager

The cache manager is the smallest of the components of the cluster. There always has to be one and only one cache manager in the cluster, as it is the component that allows the different main memory caches running in the backend nodes' node managers to collaborate and share resources effectively. It is the component that keeps a global cluster-wide registry of cached files.

ICacheManager is the only remote RMI interface exported by the cache manager process. Its two main methods are:

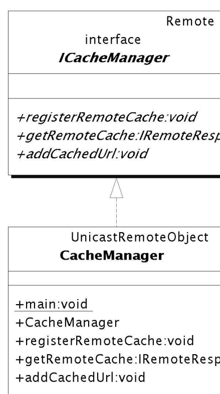


Figure 3.12: CacheManager class diagram.

registerRemoteCache Used by the node managers to register themselves with the cache manager at startup time.

addCachedUrl Notifies the cache manager that a node manager has cached a given URL. When called by node managers, the cache manager will update its mapping of URLs to node managers (or RemoteCaches) to store this new association. Every node manager should call this method of the cache manager when a new file is brought to its local cache.

getRemoteCache Returns the node manager that has the URL passed as parameter cached on its local cache, or *null* if the cache manager has no knowledge of any local cache holding this URL. The returned object is a reference to *IRemoteCache*, a remote RMI interface associated with each node manager that the querying node manager can then use to contact the other backend node's cache directly without going through the cache manager or front-endfront-end.

In the non-RMI version of the cache manager, these methods are not implemented exactly the same, but the cache manager provides the same functionality and they document well how the cache manager collaborates with node managers to keep a cluster-wide caching registry.

3.6 Node Manager

Node managers are the processes that run on the back-end nodes. Their job is to retrieve requests from the request queues associated to them at the front-end and put completed responses back in the front-end's response queue so that they can be sent to the clients. Typically only one node manager will run per machine, but this is not enforced by the software in any way.

Node managers have two alternative implementations, similar to the front-end. There is an implementation that follows a a multi-threaded model, similar to the thread-per-connection approach implemented by the MultiThreadListener, but with the difference that it handles Request objects instead of HTTP request messages sent through TCP socket connections from the Web clients. The basic approach is very similar, there is a main thread that loops forever retrieving requests and giving them to worker threads from the pool to process. When using the custom socket-based protocol, the node manager is implemented with an event-driven approach that uses a single thread for handling all the operations.

Figure 3.13: Node manager class diagram.

The main classes participating on the node manager are shown at the class diagram in Figure 3.13. At startup time, a node manager registers itself with the front-end process, establishing the communication path used between the front-end and the node manager. The front-end node can then delegate client requests to the node manager using either an

RMI-based request queue or the more efficient socket-based protocol.

3.6.1 Caching

Apart from communicating with the front-end to retrieve requests and send responses, the other main functionality included in the node managers is a main memory cache that collaborates with the local caches of the other node managers via the cache manager to provide a cluster-wide collaborative caching service.

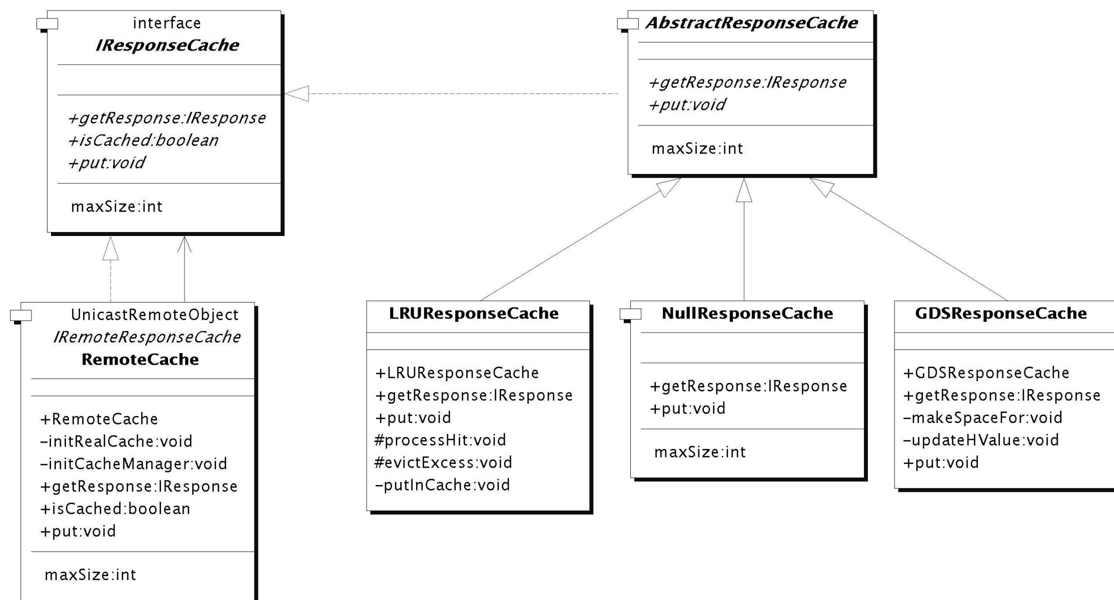


Figure 3.14: Class hierarchy for *IresponseCache*.

There are different alternative implementations of cache replacement algorithms, they all implement the *IResponseCache* interface and which one will run is chosen at node manager startup time from a command line option. A short description each one of the different concrete *IResponseCache* implementors, shown in Figure 3.14 follows:

RemoteCache The *RemoteCache* class does not implement any specific cache replacement policy by itself. It is a *facade* class that similarly to the *DispatcherFacade* discussed previously, delegates all the caching replacement decisions to another concrete implementor of *IResponseCache*.

RemoteCache deals with all the communication with the cache manager when the RMI-based protocols are used, it is the only *IResponseCache* implementation that is aware of being run in a distributed collaborative caching system. This way, the other cache replacement policy implementations are completely independent of the fact that they collaborate with other caches, and can simply focus on implementing an specific cache replacement policy as simply as possible.

When a node manager is using a *RemoteCache* as its local cache and it requests a certain response from it, the *RemoteCache* acts as a bridge between the *RemoteCache* and the rest of the node manager. The request will be delegated to the "real" local cache only if it is already in its main memory cache.

In the case that a cache miss would happen locally, the *RemoteCache* communicates with the cache manager to retrieve the response from the main memory cache of a different node manager. It will only resort to loading a response from the file-system if no other backend node contains the response/file in its main memory.

NullResponseCache This is an implementation of *IResponseCache* that does not cache anything in main memory, it will always obtain the response data reading it from the file-system each time. It is useful for testing and evaluation purposes, it is always recommendable to be able to evaluate the differences between an environment without caching and one with local or collaborative main memory caching.

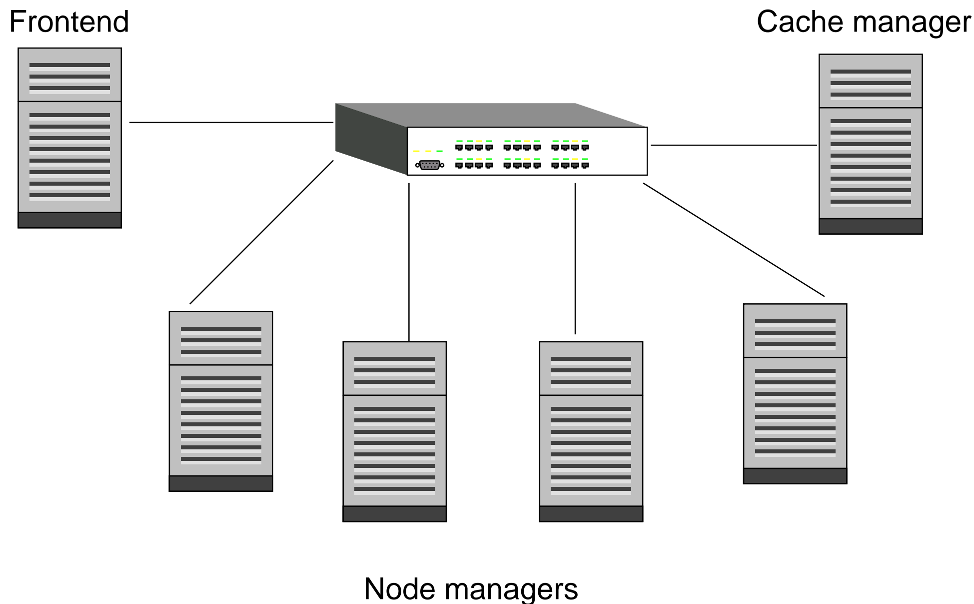
LRUResponseCache It implements the traditional Least Recently Used (LRU) replacement policy. This policy has known problems for Web caching, but it is a common standard and a good benchmark to evaluate against.

GDSResponseCache This class provides an implementation of the GreedyDual-Size replacement policy, an state-of-the-art policy that is considered the one of the best Web cache replacement policies in the research literature.

Chapter 4

Evaluation

4.1 Test setup and methodology



The cluster used for the evaluation of the Web cluster prototype has six computer connected to an Ethernet switch with a 100 MBit/second network. All the nodes are identical, with one Intel Pentium III processor at 1 GHz and 256 MB of main memory running Red hat Linux 7.2. One node is always dedicated to being the front-end and another one is always the cache manager. Depending on the test being performed, from one to four backend nodes running one node manager process each will be used.

The cluster configuration used for evaluation uses an event-driven architecture for all the components and a custom socket-based protocol for communicating between the components. The front-end uses the Replication LARD dispatcher discussed previously and the node managers' caches use the GreedyDual-Size replacement policy. Other design options discussed previously, like the RMI-based inter-process communication protocols

or multi-threaded architectures have not been used for the evaluation, as in preliminary tests they showed very poor performance and specially the RMI subsystem would run out of resources when loaded considerably.

All tests have been run with with the cluster with different numbers of backend nodes, as well as a standalone Web server. Comparing the cluster's performance and scalability with a single-node server implemented using the same Java platform allows for better understanding of the scalability issues on the cluster. The standalone Web server implements an event-driven model and sends response data using Java's new I/O facilities straight from the file-system cache of the operating system to the client. The Web server itself never handles directly or copies the bytes of the files that is serving to clients.

Each one of the available Web traces has been tested with the standalone server and the cluster in configurations of one to four backend nodes. As a single standalone Web server is already capable of saturating the available network bandwidth, a clustering approach should look into improving performance and scalability where raw file and network I/O is not the only factor. For purely static traffic, a standalone server is the best solution most of time as the results in following sections will show.

That is why the tests have all been performed twice, once with the standard Web server and cluster software and the second time with modified server-side software that introduces a CPU-intensive calculation that takes around 35 millisecond for each request handled. This modification allows the tests to simulate a request traffic in which all the requests are for dynamic program or scripts that generate the response sent to the client.

4.1.1 WebStone

WebStone [40] will be used to create the load on the Web server or cluster. WebStone works by running one or more webclient processes, one or more for each client system available for the evaluation. Each webclient makes successive HTTP requests as fast as it can receive data back from the server being tested, simulating the actions of a single very active (and fast) user visiting the site.

There is a controller process, called webmaster, that distributes the webclient software and test configuration files to the client computers. Once the test scenario is set, webmaster starts a benchmark run and waits for the webclients to report back the performance they measured. The webmaster combines the performance results from all the webclients into a single summary report.

WebStone primarily measures throughput (bytes per second) and latency (time to complete a request); it also reports pages per minute, connection rate averages and other information useful for sanity-checking the throughput results. Two types of throughput are measured by WebStone:

- Per-client throughput divides the total number of bytes by the total connection time and by the number of clients.
- Aggregate throughput is the total number of bytes transferred throughout the test divided by the total test time.

Two types of latency are reported as well:

- Connection latency represents the time taken to establish a connection.
- Request latency consists of the time taken to transfer the data once the connection has been established.

The latency perceived by the user will consist of the sum of connection and request latencies plus any latency related to the characteristics of the network connecting the client computer and the web server.

4.2 Traces

There are four set of traces that will be used for evaluating the Web cluster software:

WebStone The standard file-set and file-list that comes with WebStone [40], it is a very small file-set that does not represent typical Web traffic but is useful for comparing with the other more "real-world" traces.

NASA Based on traces from the Web server at NASA's Kennedy Space Center [37].

Calgary Based on traces from the Department of Computer Science at the University of Calgary [37].

USASK Based on traces from a campus-wide Web server at the University of Saskatchewan [37].

For the last three traces, which are publicly available, the data has been reduced to include only GET requests, and simplified so that it can be used for driving WebStone. The next table summarises the main characteristics of the different traces as they are used for driving WebStone:

	NASA	Calgary	USASK	WebStone
Total Requests	**1,692,590**	**565,513**	**1,024,864**	**1,090**
Total Bytes Transferred (MB)	**86,505**	**18,089**	**4,783**	**18**
Mean Transfer Size (bytes)	**53,590**	**33,541**	**4,894**	**19,753**
Working Set Size (MB)	**258**	**321**	**216**	**5.54**

4.3 Evaluation results

The tests that have been carried out have been all driving WebStone with the different requests from the traces mentioned previously. For each run with a given trace three metrics are collected:

Connections per second How many HTTP connections the server has been able to handle per second.

Throughput The number of MBits per second that the server has served during the test.

Average response times How many seconds on average took the server to fulfill a request during the test.

Furthermore, the test results have been separated depending if they are obtained with static or dynamic traffic. The static traffic is driven by the four different traces alone and the dynamic traffic is generated by running WebStone with the same traces, but with a modified version of the Web serving software that introduces a processing overhead for each request to simulate dynamic request handling.

In the graphs that follow in this section the x axis goes from zero to four, and represents the number of cluster backend nodes running node managers that are used for each test. That is, the cluster will always be running with one system as the front-end, another one as the cache manager and from one to four nodes running one node manager process each one. When the number of backend nodes is zero, it means that the standalone event-driven Web server discussed previously has been used for that test, in this case only one machine handles all the requests.

Figures 4.1 and 4.2 show the connections per second obtained with dynamic and static traffic respectively. When dynamic requests are used, the cluster shows good scalability, increasing connection rate significantly when backend nodes are added to the configuration. For all traces except the NASA one, the connections per second suffer a small decrease when comparing the standalone server with a one-backend cluster configuration, and the rate obtained with a four-backend cluster is over 300% higher than with the standalone server.

Connection rate results are very different when static requests are used (Figure 4.2), the standalone server always outperforms the cluster in this case. The cluster configurations obtain a flat connection rate with minimal variations when backend nodes are added, this rate is between a 14% and 70% of the connections per second of the standalone server, depending on the trace.

Figures 4.3 and 4.4 represent the throughput results with dynamic and static requests respectively. With dynamic requests the pattern is very similar to the connection rates discussed above, the cluster achieves better throughput than the standalone server when

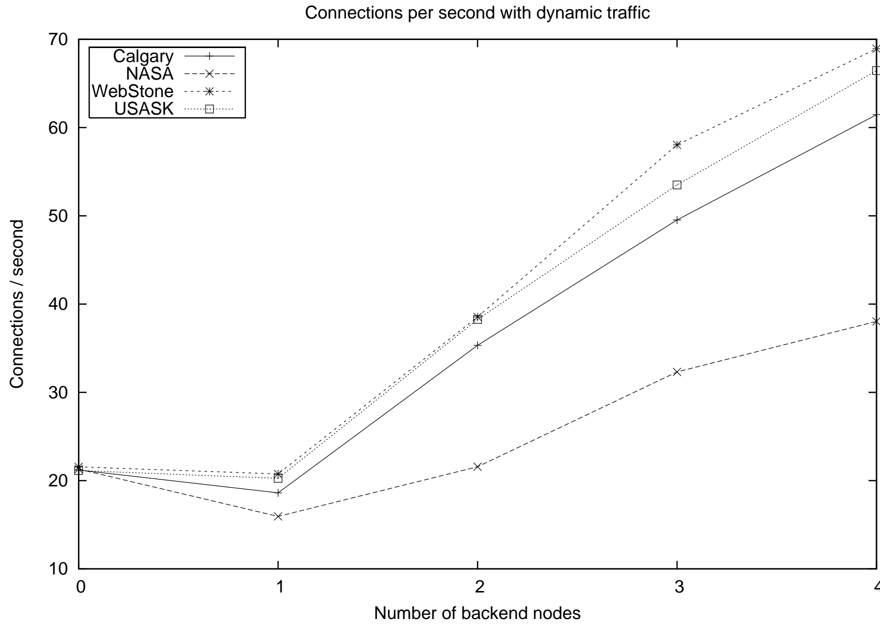


Figure 4.1: Connections per second with dynamic traffic.

backend nodes are added, with a throughput decrease when the one-backend cluster is used and increasing as more backend nodes are added.

Throughput results for static requests are in Figure 4.4. In this case, throughput decreases considerably when comparing the standalone server with a one-backend cluster, and stays flat or increases slightly as backend nodes are added. It is important to notice that in the case of the WebStone and NASA traces the standalone server is already saturating the network, obtaining between 80 and 90 Mbit/second on a 100 Mbit/second Ethernet network. This should be very close to the maximum throughput possible with these traces, given that there is a TCP overhead for each request.

The third and last metric collected in the tests is the average response time that takes to reply a given request, an important aspect to measure the performance of a Web server from the users' point of view. These results are shown in Figures 4.5 and 4.6. As it has happened with connection rate and throughput, the results are very different depending if dynamic or static traffic is evaluated.

With dynamic traffic (Figure 4.5), the cluster achieves considerably better (lower) response times than the standalone server when more than one backend node is used. The single-backend cluster configuration has slightly higher response times than the standalone server, but they decrease rapidly as backend nodes are added. When considering static traffic (Figure 4.6), the behaviour is that response times increase when going from the standalone server to the one-node backend cluster but they only decrease slightly when more backend nodes are used, staying always higher than the standalone server's response times.

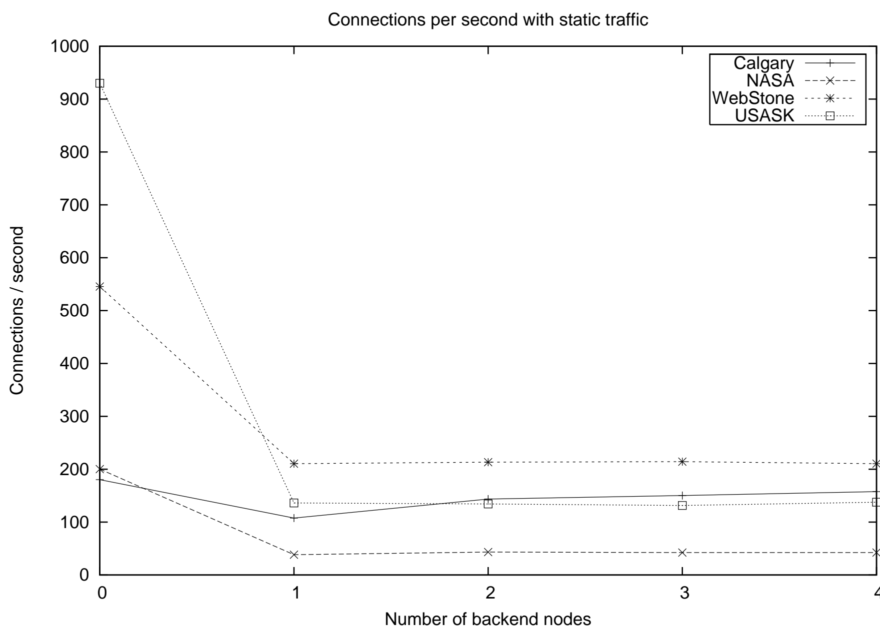


Figure 4.2: Connections per second with static traffic.

After looking at all the test results, it is clear that the static and dynamic workloads have very different effects on the scalability of the cluster. When all the requests are for static files that require the server to simply obtain the contents of the file and send it to the Web client, the work of the server is I/O intensive, and a Web server running on a single machine is able saturate a 100 Mbit/second network without using any caching. This Web server can be implemented in Java, and be able to handle hundreds of connections per second easily using an event-driven design that avoids threading overhead and directly transfers data from the file-system cache to the client socket without excessive copying.

When the traffic is mainly composed of static requests, it is very hard for a single-front-end based Web cluster to improve the performance obtained with a single-node Web server. In a cluster with systems interconnected with 100BaseT Ethernet, pure I/O performance will always be lower than for a single-node Web server, as both request and response data has to be copied and sent between the nodes. For this kind of traffic, a single-node Web server with a good I/O subsystem will outperform a cluster for most workloads. The cluster's data path is too long and hits a bottleneck very early when its workload is I/O intensive. For extremely high workloads and static traffic, replicated servers in different locations are probably preferable.

Results are very different for dynamic traffic, here the cluster with more than one backend node is able to outperform the single-node Web server in all metrics considered. Dynamic requests stress the Web serving system in a very different way than static requests. While for static requests pure I/O throughput of the system is the most important factor, dynamic traffic uses the Web server's CPU much more, and can benefit from the aggregated processing power available in a Web cluster. Performance of the standalone

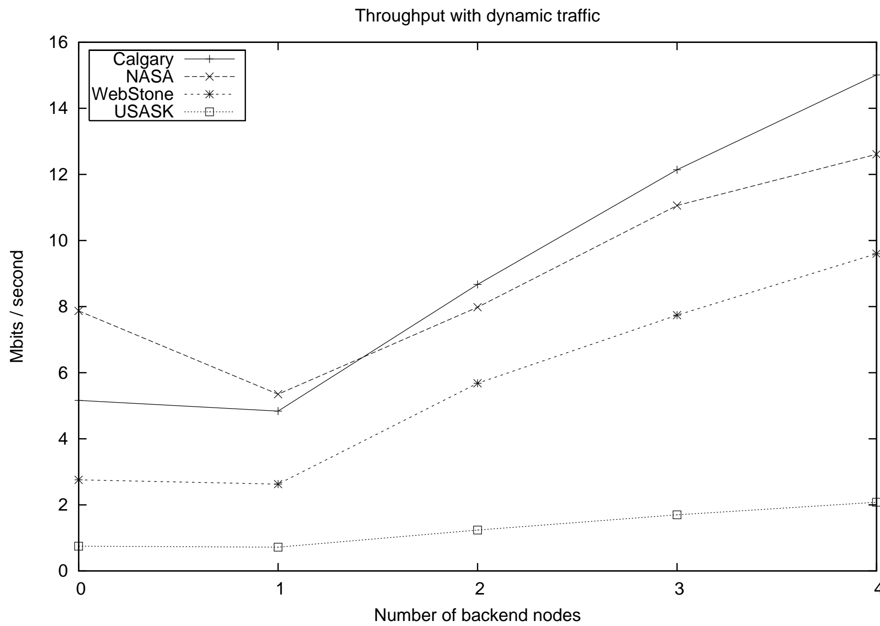


Figure 4.3: Throughput with dynamic traffic.

Web server is comparable or slightly higher than the one-backend cluster for dynamic traffic, but the cluster's performance increases significantly as backend nodes are added, providing much higher scalability for this kind of workload.

Handling static and dynamic workloads require very different qualities from Web servers, static traffic handling is I/O-bound while dynamic traffic is more CPU-bound than I/O-bound. In I/O-bound applications optimising the path of data and avoiding data copying are very important, and a Web cluster that uses a single front-end node and standard Ethernet network interconnections performs poorly in those aspects. However, when the majority of the traffic is dynamic and thus CPU-bound, the cluster's limitations from I/O processing performance point of view are a less important factor than the benefits obtained from the aggregated CPU power of the cluster.

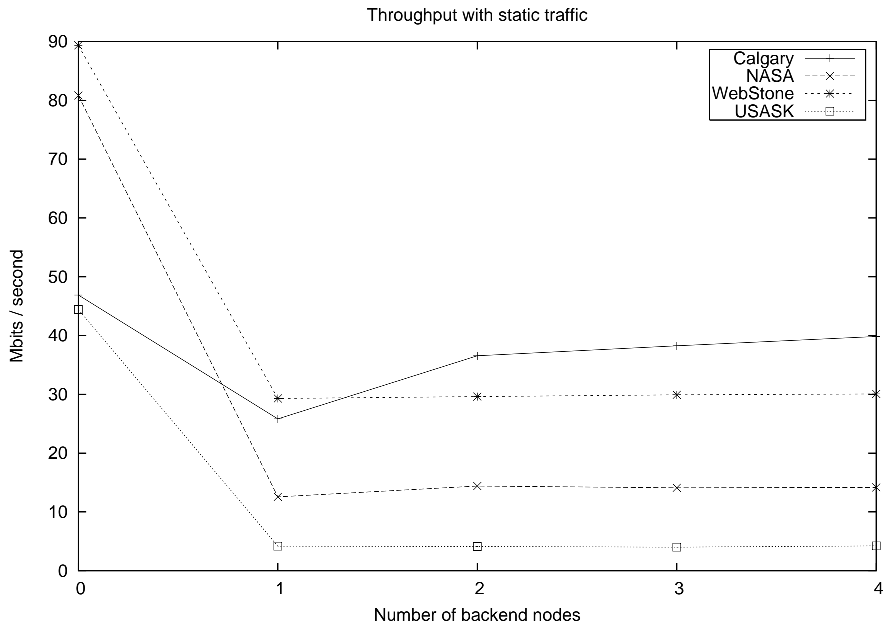


Figure 4.4: Throughput with static traffic.

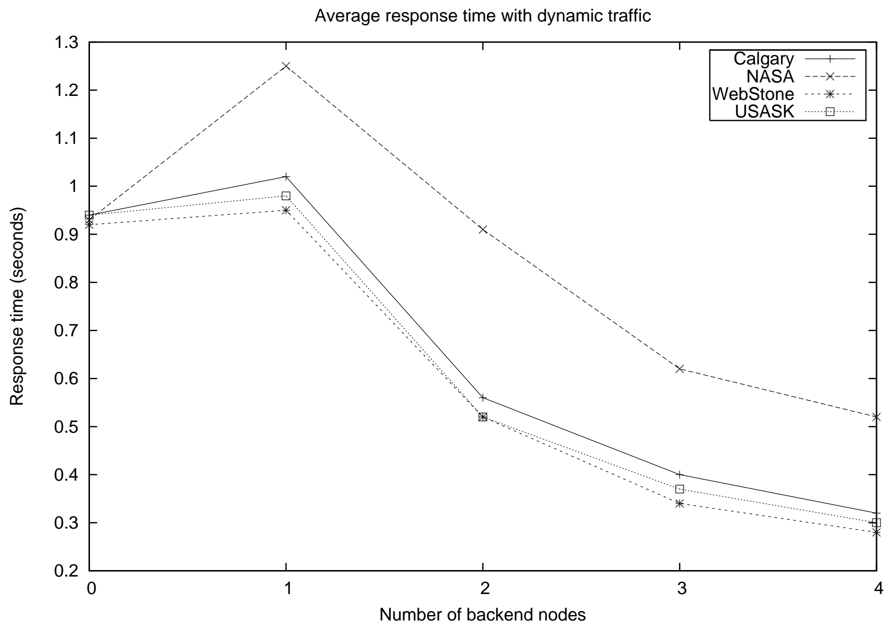


Figure 4.5: Average response times for dynamic traffic.

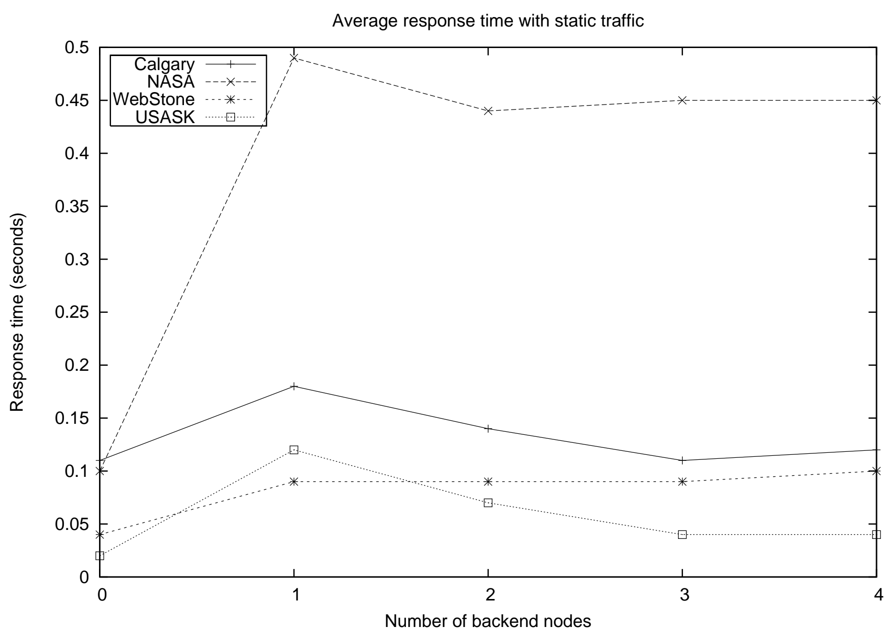


Figure 4.6: Average response times for static traffic.

Chapter 5

Conclusions and future work

Cluster Web servers are a popular architecture used for improving the scalability of high-traffic Web sites. Although state-of-the-art commercial Web cluster front-ends offer advanced functionality such as content-aware or layer 7 request distribution, their architecture is based on the requirements of much simpler layer 4 load-balancers that do not need to perform sophisticated processing of the request data.

The scalability offered by dedicated front-end nodes is limited by their lack of integration with the back-end nodes' software, which is one of their selling points as they require minimum modification or reconfiguration of the back-end Web server software. Thus most clustering products don't work as an integrated whole, they are implemented as separate front-ends or Web switches that "clusterise" a set of back-end Web server nodes.

In this dissertation a Web cluster architecture is designed, implemented and evaluated. With the use of COTS hardware and integrated front-end and back-end software better scalability and flexibility should be achieved for Web clusters. Furthermore, the clustering software is all implemented in Java, with the goal of evaluating Java as a platform for the implementation of high-performance network servers. The Web cluster prototype described here is built out of a set of distributed components that collaborate to achieve good scalability as a Web server. As all the components have been designed from scratch to integrate with each other and work well together, the scalability problems associated with state-of-the-art content-aware cluster front-ends are avoided.

The clustered Web server described in the dissertation includes advanced algorithms for request distribution and caching, implementing a cluster-wide caching strategy that allows for optimal management of main memory caches in all the nodes in the cluster. The cluster prototype has been implemented using Java, a platform that is not a common choice for the development high-performance network servers, but has shown to scale well. Design alternatives for highly-scalable Java servers have been discussed, especially relevant here are the new Java 1.4 I/O facilities and their evaluation for the implementation of scalable event-driven network servers.

The non-blocking I/O facilities available since Java 1.4 allow for truly scalable systems to be implemented in Java, but there is still a lack of documented best practices and design patterns that show developers how to take advantage of them. Moreover, the non-blocking I/O APIs are not well integrated with the rest of the Java I/O APIs, complicating the modification of existing systems that wish to use them. Most frequently applications will need to be redesigned to adopt an event-driven architecture so that they can achieve the scalability benefits of the non-blocking I/O facilities. This is not an easy task as most Java server applications are currently based on a thread-per-connection model, an approach that is very different to the event-driven model. Another issue related to the new Java I/O APIs is that they can manage and allocate memory out of the Java virtual machine heap, which makes it possible for the application to use too much memory without the direct control of the developer or JVM, which can eventually cause the server process to be stopped by the operating system.

Although the Java platform does include all the functionality needed for implementing highly-scalable network servers, they are bundled together with legacy APIs that are not suitable for high performance uses. RMI is an example of a component of the standard Java platform that is implemented in a completely non-scalable way. More work is needed to re-implement RMI taking into account scalability concerns or at least providing alternative implementations suitable for high-performance applications. Similarly, Java still has extensive blocking I/O APIs that haven't been integrated well with the new I/O facilities, making it difficult to evolve existing server applications toward using non-blocking I/O features.

The Web cluster prototype has been evaluated using a trace-driven Web benchmarking tool, WebStone [40], and Web server access logs that are publicly available. Two kinds of traffic have been simulated for each trace and test, static and dynamic. Static traffic is comprised of normal HTTP requests, taken directly from Web server access-logs. Dynamic traffic refers to requests that are directed to some kind of Web application that generates the responses for each request. These dynamic requests have been simulated by modifying the Web server to perform a calculation that takes approximately 35 milliseconds for each request. Previous Web clustering research has focused more on static traffic. The introduction of a workload with dynamic requests is a key aspect of the evaluation, as dynamic Web sites are very common nowadays and the tendency is toward mostly dynamic sites in the future.

The evaluation results show that a differentiation between static and dynamic requests is necessary for a better understanding of Web server scalability issues. When handling static traffic, it is easy for a single node Web server to saturate an Ethernet 100 Mbit/second connection, and I/O throughput is the main factor. However, scalability of Web servers that handle dynamic traffic is much improved from using a clustering ap-

proach. This is due to the fact that when serving dynamic requests I/O performance is not the main scalability factor and CPU processing becomes very important as well. Furthermore, issues involved in implementing scalable network servers in Java are discussed, focusing specifically on how the new non-blocking I/O facilities affect the architecture and scalability of server applications.

A single-node standalone Java Web server that uses an event-driven architecture has been implemented for comparing its scalability with the Web cluster's. This server has been able to outperform the cluster prototype for most of the metrics measured with the static workload. It is clear that when considering Web traffic that is comprised of mostly static requests to files in the server, the overhead and data copying involved in a cluster offers no substantial advantage over a high-performance standalone Web server for most loads. A single node standalone server has a shorter data path and can offer better I/O throughput than a set of cluster nodes interconnected by a FastEthernet network.

However, if dynamic traffic is considered the advantages of a clustered architecture are clear. When sheer I/O throughput is not the only factor the cluster prototype outperforms the standalone Web server in all the tests evaluated. When requests involve some kind of processing that requires the use of the CPU, a cluster can provide better scalability due to the aggregated CPU power of the back-end nodes, while the standalone node is limited to its single CPU (in the case of the nodes in our test network). Thus, in the transition from Web serving to Web *application* serving, real scalability advantages can be obtained from using Web clusters instead of standalone Web servers.

For Web workloads that are still mainly static and need massive scalability, there is a need for more research on distributed dispatchers with Web serving platforms. For I/O bound applications, a single dispatcher becomes a bottleneck, and it could be interesting to see how architectures like Scalable LARD [4] could be applied to enable clustering in modern Web and application servers.

It is probable that static Web sites will lose popularity compared to dynamic sites in the future, so more research is needed to see how Web application platforms can benefit from clustering architectures. In the case of the Java platform the reference implementation of the Java Servlet specification, Jakarta Tomcat [36], could be used as a starting point for more research on clustering applied to Web application servers. JBoss[15] is a freely available Java application server that has included clustering functionality in recent versions.

Bibliography

[1] Ludmila Cherkasova and Gianfranco Ciardo. Role of Aging, Frequency, and Size in Web Cache Replacement Policies.

[2] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In Proceedings of sthe USENIX Symposium on Internet Technologies and Systems, pages 49–60, December 1997.

[3] Marc Abrams et. al. Caching Proxies: Limitations and Potentials. Technical Report, Department of Computer Science, Virginia Polytechnic University and State University, July 17, 1995.

[4] Mohit Aron, Darren Sanders, Peter Druschel and Willy Zwaenepoel. Scalable Content-Aware Request Distribution in Cluster-based Network Servers. Proceedings of the USENIX Symposium on Internet Technologies and Systems 2000.

[5] Cisco Systems, `http://www.cisco.com`.

[6] Doug Lea. Concurrent Programming in Java, Second Edition. Addison Wesley Longman, November 2000.

[7] Mohammad Salimullah Raunak. A Survey of Cooperative Caching, December 15, 1999.

[8] Alec Wolman et. al. On the Scale and Performance of Cooperative Web Proxy Caching. Published as *Operative Systems Review* 34, December 1999.

[9] Chu-Sing Yang and Mon-Yen Luo. Efficient Support for Content-based Routing in Web Server Clusters. In Proceedings of USITS'99.

[10] A, Shaikh, R. Tewari and M. Agrawal. On the Effectiveness of DNS-based Server Selection. IBM Research Report, Proceedings of IEEE Infocom, April 2001.

[11] Foundry Networks. `http://www.foundrynet.com`.

[12] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithm. Proceedings of the USENIX Symposium on Internet Technologies and Systems 1997.

[13] Ludmila Cherkasova. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy, HP Computer Systems Laboratory 1998.

[14] David Mosberger and Tai Jin. httperf - A Tool for Measuring Web Server Performance. http://www.hpl.hp.com/personal/David_Mosberger/httperf.html.

[15] JBoss Java application server. http://www.jboss.org.

[16] AJ. Andersson, S. Weber, E. Cecchet, C. Jensen and V. Cahill. Kaffemik - A distributed JVM in a single address space architecture. SCI Europe 2001 Conference.

[17] Vivek S. Pai et. al. Locality-Aware Request Distribution in Cluster-based Network Servers. Proceedings of the eighth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[18] T. Larkin. An Evaluation of Caching Strategies for Clustered Web Servers, (MSc Dissertation), November 2001.

[19] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. University College London CS Research Note RN/98/13.

[20] Xuehong Gan, Trevor Schroeder, Steve Goddard, and Byrav Ramamurthy. Highly Available and Scalable Cluster-based Web Servers. In submission to The Eighth IEEE International Conference on Computer Communications and Networks, October 1999.

[21] Wensong Zhang. Linux Virtual Server For Scalable Network Services. http://www.LinuxVirtualServer.org.

[22] Eric Anderson, Dave Patterson and Eric Brewer. The MagicRouter, an Application of Fast Packet Interposing. Submitted for publication in the Second Symposium on Operating Systems Design and Implementation.

[23] Nortel Networks, http://www.nortelnetworks.com.

[24] Om. P. Damani et. al. ONE-IP: Techniques for Hosting a Service on a Cluster of Machines. Proceedings of the Sixth International World Wide Web Conference.

[25] Douglas Schmidt et al. Patter-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects. John Wiley & Sons Ltd., April 2001.

[26] Emanuel Cecchet. Parallel Pull-Based LRU: a Request Distribution Algorithm for Clustered Web Caches using a DSM for Memory Mapped Networks. Third International Workshop on Software Distributed Shared Memory (WSDSM'01) in Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'2001) Brisbane, Australia - May15-18, 2001.

[27] Roland P. Wooster and Marc Abrams Network Research Group, Computer Science Department, Virginia Tech, 1997. http://research.cs.vt.edu/chitra/docs/www6r.

[28] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla and Edward A. Fox. Removal Policies in Network Caches for World-Wide Web Documents, 1996.

[29] Resonate, http://www.resonate.com.

[30] P. Srisuresh and D. Load Sharing using IP Network Address Translation (RFC 2391), August 1998.

[31] T. Brisco. DNS Support for Load Balancing, RFC 1794. April 1995. http://www.rfc-editor.org/rfc/rfc1794.txt.

[32] Steve Goddard and Trevor Schroeder. The SASHA Architecture for Network-Clustered Web Servers. In proceedings of the 6th IEEE International Symposium on High Assurance Systems Engineering, 2001.

[33] T. Schroeder, S. Goddard and B. Ramamurthy. Scalable web server clustering technologies. IEEE Network, May/June 2000.

[34] Web caches design for SCI clusters using a Distributed Shared Memory C. Perrin and E. Cecchet Second Workshop on "Parallel Computing for Irregular Applications WPCIA2” Toulouse, France - January 8, 2000.

[35] David A. Maltz and Pravin Bhagwat. TCP Splicing for Application Layer Proxy Performance. IBM Technical Report RC 21139.

[36] Apache Software Foundation, Jakarta Project. http://jakarta.apache.org.

[37] Martin F. Arlitt and Carey L. Williamson. Web Server Workload Characterization: The Search for Invariants. 1996 ACM SIGMETRICS Conference, Philadelphia, PA, May 1996.

[38] Ariel Cohen, Sampath Rangarajan and Hamilton Slye. On the Performance of TCP Splicing for URL-aware Redirection, Proceedings of USITS 1999, The 2^{nd}USENIX Symposium on Internet Technologies & Systems.

[39] Ludmila Cherkasova and Magnus Karlsson. Scalable Web Server Cluster Design with Workload-Aware Request Distribution Strategy WARD, Computer Systems and Technology Laboratory, HP Laboratories Palo Alto, July 2001.

[40] WebStone, by Mindcraft. http://www.mindcraft.com.