Single System Image Servers on top of Clusters of PCs

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Vlad Olaru

aus Bukarest, Rumänien

Tag der mündlichen Prüfung:	3. Dezember 2004
Erster Gutachter:	Prof. Dr. Walter F. Tichy
Zweiter Gutachter:	Prof. Dr. Martina Zitterbart

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The clusters of COTS (Commodity Off The Shelf) computers have shown lately a great potential for high performance computing as they offer price-competitive and scalable solutions (huge aggregate main memory, price-competitive, highly available and scalable secondary memory, huge I/O bandwidth). However, as of now, most of these clusters are still regarded as a collection of independent machines *explicitly* cooperating to some extent in order to fulfill some task. In spite of the tremendous potential of the hardware interconnects they are equipped with, the cluster systems fail today to present a Single System Image (SSI) to the user. Indeed, nowadays high-speed System Area Networks (SAN) have latency and bandwidth figures comparable to those of memory subsystems, and, thus, advocate for a tighter integration of the various resources in the cluster. Such an integration requires however appropriate mechanisms and management policies operating across the cluster. Traditionally, such tasks are reserved to the operating system. Currently however, there are no state-of-the-art cluster operating systems available. Moreover, the existing stand-alone operating systems offer conventional services that are a poor match to the expectations of the parallel and/or distributed applications run on top of the clusters. Therefore, most of the time, the application developers implement these services in user-space. As a direct result, the overall performance of the system may suffer, while the software complexity of the applications increases substantially (with adverse effects on maintenance and further development).

There is a wealth of previous research results that encourage us to follow the idea of a cluster operating system. On the system side, a first step towards a tighter resource collaboration was the development of scalable user-space communication subsystems [68, 27, 70, 66]. They aimed at a reduced communication latency by removing the operating system from the critical path. The reasons behind the

OS bypass were the unwanted performance penalties induced by double buffering and some kernel specific mechanisms (such as context-switching). However, since message passing is not the most handy programming model, the next step taken was to develop higher level software abstractions (memory pages, disk blocks, etc.). The most notable research effort in this direction was that of the software Distributed Shared Memory (DSM) systems [3, 76].

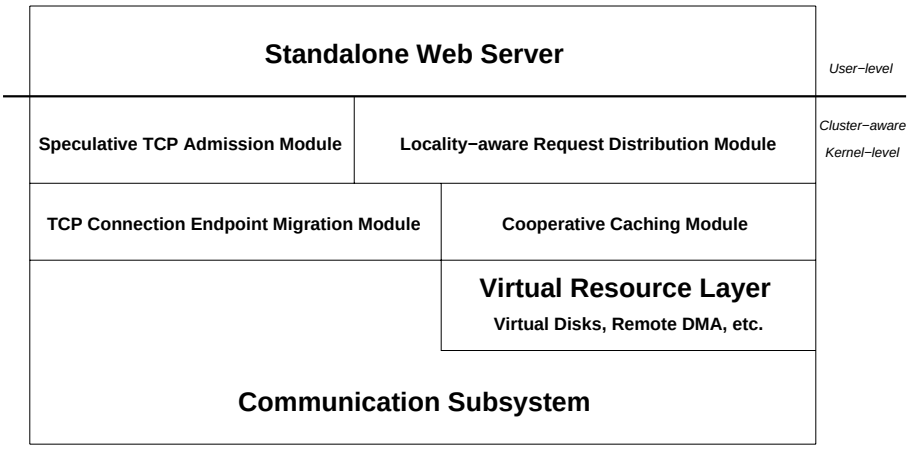


Figure 1.1: A possible software architecture of a cluster operating system

The low latency of the emerging high-speed networks as well as the early experience with software DSM systems lead at some point to a significant twist in the traditional approach to the parallel/distributed file system memory hierarchy (server disk, server cache and local client cache). The cooperative caching network file systems [23, 4] allowed checking the local cache misses also against the remote client caches before going to the server cache. The low-latency networks play a paramount role here, as retrieving remotely cached blocks is faster than getting them from the server disk, even if the disk happens to be local. Moreover, as in the case of DSMs, by implementing a joint management of the cluster caches, cooperative caching enables the working set to scale beyond the limit of the locally available memory.

A third notable trend, orthogonal to the above issues, made a case for flexibility/extensibility in the kernel operation [14, 28]. The conventional general-purpose kernels respond inadequately to the challenges imposed by the new class of the highly I/O-bound applications (mostly related to the multimedia and Web/Internet technology domains). Systems like those cited above use a joint management of the system's resources between the applications and the operating system. These kernels let the applications manage their own resources, while the system software

continues to provide general mechanisms such as protection domains, resource allocation and scheduling, etc.

Summarizing some of the above ideas, Figure 1.1 depicts a possible software architecture of a cluster operating system for Single System Image Servers on top of COTS clusters. At the lowest level, a message passing subsystem enables the communication over the SAN. The next level virtualizes the accesses to the remote resources in order to provide higher level abstractions than those of simple messages. Above the Virtual Resource layer, one can build entire kernel subsystems, but in this thesis we will restrict our attention to the cooperative caching, TCP connection endpoint migration, locality-aware request distribution and speculative TCP admission modules. At the top level, an unmodified stand-alone Web server program runs on the distributed infrastructure of the cluster as if it would do on a single machine. Such an approach is made possible by using Single System Image cluster operating system services that are provided by the aforementioned modules. Explaining how all these modules work together in order to provide for a Single System Image of the cluster-based server is the topic of this thesis.

1.1 Definitions

Before we proceed any further, we take the opportunity to define some of the main concepts for this thesis.

Single System Image Servers (SSI) - A Single System Image Server on top of COTS clusters is a software construct that hides the distributed nature of the hardware infrastructure of the cluster and presents both the system developer and the user with the image of a single virtual server with multiple processors operating like a stand-alone, single machine server would do.

Cooperative caching - A cooperative caching system manages jointly the individual file system caches in a cluster and offers support for a global, unified, cluster-wide cache.

Exclusive caching - Exclusive caching is a technique that avoids storing multiple copies of the same disk block either in the various caches of the same computing system or among the distributed caches of several computers.

TCP connection endpoint migration - The TCP connection endpoint migration allows a client-transparent, arbitrary assignment and reassignment of server-side TCP connection endpoints to particular server nodes in a cluster.

Locality-aware request distribution - A locality-aware request distribution system for cluster-based servers attempts to provide better amortized performance by routing requests to individual server nodes according to the locality of the requested data.

1.2 System overview

The purpose of this thesis is to investigate the impact of integrating various cluster resources into a Single System Image server. Of major concern is not only the performance of the system but also its ease of use and programming, its flexibility and the capability to offer global high-level abstractions/services that hide the distributed nature of the server and the underlying message passing based infrastructure.

Today, big "computer farms" are common place in the server industry. Front-end computers are used to redirect requests to servicing (back-end) computers. The request distribution is mostly done in conjunction with some back-end load balancing policy. A lot of work has been done in the area of load balancing the back-end machines, but some of the previously mentioned features of the COTS clusters using SANs suggest possible benefits coming from an enhanced cooperation among the back-end servers. The typical server architecture that we will be referring to throughout this thesis is depicted in Figure 1.2.

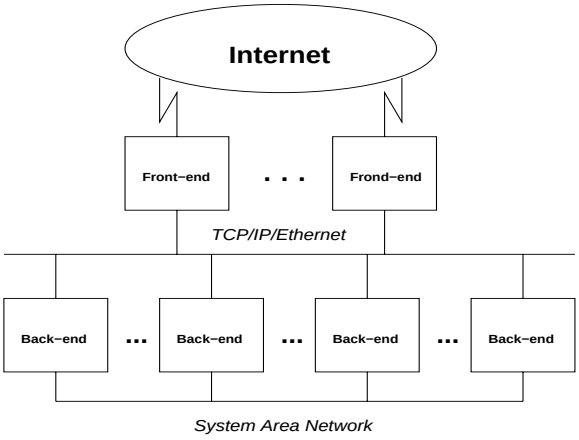


Figure 1.2: Server architecture

Such a cooperation may involve changes in the server applications, but getting good performance out of such constructs would be possible only by changing the operating systems as well. Since the traditional kernels have been developed for a stand-alone run, they fail to match adequately the challenges imposed by a distributed server. However, only a collaboration between the user-level application (in our case, the server program) and the kernel would yield the best performance, as the general purpose kernel algorithms for memory management, process scheduling, etc. respond inadequately to specific application needs.

Our approach to integrating various resources into a single server system con-

cerns two major parts of the operating system: the disk and the network subsystems. The main mechanisms that we developed are the Cluster-Aware Remote Disk drivers (CARDs) and the TCP connection endpoint migration. They serve as driving engines for the cooperative caching and locality-aware request distribution policies.

The CARDs are block oriented drivers that virtualize remote disk accesses over a SAN. They offer a higher level of abstraction (I/O pages or disk blocks) to the user/developer than that of the message passing systems. Through exclusive caching, the CARDs exercise cluster-wide a fine-grain control over multiple cached copies of a block. The exclusive caching is made possible by a single copy protocol that allows performing DMA from the disk to the network buffering system. The operation of the CARD drivers may be driven by cooperative caching policies. A set of CARDs implementing a common cooperative caching policy builds a unified disk cache across the cluster. Each such set may employ its own policy, independently of other similar sets. A flexible design enables the applications to download at will their favorite policy into the kernel. We designed a decentralized policy called Home-based Server-less Cooperative Caching (HSCC). We build upon HSCC in order to perform request distribution-aware caching in cluster-based Web servers. Thus, the kernel software can benefit from application-level knowledge. Both the CARD drivers and HSCC have been implemented as Linux kernel modules.

The TCP connection endpoint migration is a flexible way of assigning and reassigning server-side TCP connection endpoints to particular back-end machines. Therefore, it represents a connection routing mechanism that helps build back-end level request distribution algorithms. It hides from the client the distributed nature of the server, for the client sees only a generic server-side endpoint to which it connects, irrespective of its actual physical server binding. The connection endpoint migration comes in two flavors: using front-ends and totally decentralized (actually, the back-ends take over the responsibilities of the front-ends). The mechanism can be downloaded into the kernel as a Linux module. Through speculative TCP connection admission, a fully distributed (i.e., back-end level) load balancing mechanism is possible. The speculative admission builds upon the connection endpoint migration by accepting incoming connections on overloaded cluster nodes only to offload them onto lighter loaded nodes.

We tested our system by using a popular stand-alone (i.e., non-distributed) Web server program, Apache [5], and both commercial and academic benchmarks (WebStone [61] and S-Clients [10], respectively). We transparently ran Apache instances on each cluster node, without any modifications to the server program code. Only the underlying kernels were aware of the inter-node cooperation, while the user-level daemon serviced requests as it would do on a single machine.

1.3 Contributions

This thesis proposes and proves the following claims:

Claim 1 *System Area Networks support performant implementations of higher level cluster operating system abstractions than message passing systems.*

While the performance figures of the SANs are one of the main incentives behind our work, it is an open question whether these figures translate directly into similar performance figures for constructs implementing higher-level cluster operating system abstractions (i.e., virtual disks and I/O pages or disk blocks) than message passing systems, as such constructs depend significantly not only on the performance of the communication infrastructure but also on the performance of the stand-alone operating system (network interrupt handling latency, copying between various buffering systems inside the kernel, context-switch sensitivity, fairness with respect to scheduling, etc.). Our CARD drivers help us figure out these issues.

Claim 2 *The performance of the virtual disk drivers depends significantly on the degree of asynchrony of the virtual disk driver implementation.*

Because the virtual disks fetch remote data over SANs, their design is highly sensitive to the model of computation chosen to handle the remote requests: event-driven vs. blocking, interrupt-time vs. kernel-thread, asynchronous vs. synchronous. We show that a highly asynchronous mixture of interrupt-time and kernel thread processing ensures an optimal performance for our CARD drivers while maintaining a certain degree of fairness.

Claim 3 *Exclusive caching as a fine-grain control over multiple cached copies of a block across the cluster can be implemented through a single copy remote fetch protocol.*

The current kernel design of the disk drivers requires all the disk accesses to go through a specialized cache called the buffer cache. Even the direct disk accesses use the same data structures, but, as soon as the data is delivered to the user, no copy of it is kept in the buffer cache. When it comes to virtual disks, a similar problem arises. A request for a remote disk block may require a direct access to the remote disk in order to avoid leaving behind a block copy in the remote buffer cache. This fine-grain control over remote copies is a form of exclusive caching [72, 19], and is implemented in our system [49] by means of a single copy protocol. This protocol enables the remote disk nodes to perform DMA from the disk driver into the socket buffers of the networking subsystem used by the SAN card.

Claim 4 *Cooperative caching is an effective and efficient Single System Image cluster construct only when taking into account additional parameters such as the loads of the cluster nodes.*

Exploring the performance of cooperative caching as a Single System Image

service for COTS clusters includes a thorough analysis of the mechanism itself. Cooperative caching is an appealing idea because the remote memories accessed over the SAN have better latency figures than the disks (even if the disks are local) and because the global cooperative cache it builds can accommodate larger working sets. However, certain aspects of the cooperative caching algorithms need to be addressed carefully. For instance, with our Home-based Server-less Cooperative Caching algorithm [48], we show that saving locally evicted blocks into remote memories doesn't pay off if the target nodes are chosen irrespective of their loads. For heavy workloads, most of the cache hits are global hits (i.e., hits in the remote caches), and, thus, saving locally evicted blocks should be done with care, as due to the access patterns of the applications most of the saving work may prove useless.

Claim 5 *Using cooperative caching for cluster-based servers is an effective alternative to replication.*

Using cooperative caching in cluster-based servers has been explored to a smaller extent. Moreover, the solutions that we are aware of relied on simulation. We used the CARD drivers and Home-based Server-less Cooperative Caching to show that cooperative caching improves the performance of a Web system storing its data on virtual disks by making up for the performance gap between this solution and one using replication. This result is important especially for COTS clusters that rely on virtual disks to split the data center from the processing unit for administration reasons (maintenance, fault-tolerance, etc.).

Claim 6 *The TCP connection endpoint migration can be implemented efficiently as a means for back-end level request distribution in cluster-based servers.*

The very few client-aware (or client-server) TCP connection migration mechanisms designed and developed so far have been used mostly for other purposes than request distribution: to approach host mobility [59], as a fail-over mechanism for switching between an unresponsive server and a more responsive one [60], or as a driving engine behind server session migrations [63]. We show that the TCP connection endpoint (client-transparent, server-side only) migration performs well for back-end level request distribution too. For instance, simple locality-aware request distribution policies migrating non-persistent HTTP connections for given classes of static Web documents outperform clearly the Round Robin policies operating on a similar setup. Such policies compare favorably to Round Robin for persistent HTTP connections as well. The need for back-end level request distribution mechanisms is supported also by previous research results [7].

Claim 7 *The speculative TCP connection admission is an effective load balancing mechanism at the back-end level.*

The speculative TCP connection admission is a mechanism that accepts speculatively incoming connection requests only to offload them subsequently onto lighter-loaded nodes by means of TCP connection endpoint migration. It can be

used as a back-end level load balancing mechanism that offers independence of the external context to the SSI cluster-based servers (in particular, independence of the various front-ends/switches performing connection routing according to various load balancing and/or content-aware policies). Addressing the load balancing issue at the back-end level has also the advantage of a smoother integration with locality-aware request distribution policies, as the locality information resides naturally at the back-end level. We show that the speculative TCP connection admission offsets effectively the cluster imbalances induced by suboptimal request routing decisions taken outside the cluster (either at the front-end level or at earlier stages in multi-tier server architectures).

Claim 8 *The cluster-wide SSI kernel services are simplifying the cluster programming and use.*

The clusters have known widespread acceptance and success, but little has been done to adjust the kernels to the cluster operation. Most of the time, the distributed/parallel applications run on top of the traditional kernels are forced to deal either with inappropriate abstractions or with unperformant services. The virtual disks offer the cluster applications high level software abstractions with performant access that help circumvent the difficulties of programming message passing systems. For instance, the nodes mount remote disks as local storage by means of CARD drivers and that makes the remote disk accesses appear as if issued to local disks. If cooperative caching policies are in use, they even benefit of an extended cache (the global cache). By using TCP connection endpoint migration protocols in cluster-based servers, the logical equivalence of the back-end nodes gets system level support as all the back-ends are now capable of handling a given server-side connection endpoint irrespective of its physical server binding. The cluster-wide request dispatching policies benefit from the back-end level request migration support and have to implement only the heuristic that triggers the migration.

Claim 9 *The cluster-wide Single System Image kernel services allow a transparent use of unmodified stand-alone applications in a distributed/parallel environment.*

One important aspect that should not be forgotten is the wealth of stand-alone applications. Many of these have established themselves as first hand solutions for a specific problem (e.g., Web server programs like Apache [5]). It is unreasonable to hope that the new emerging distributed kernels will be adopted easily by the applications developers or will determine an immediate response to their new service offers. It is therefore imperative to make the old stand-alone applications runnable on top of the new distributed kernels by using implicitly the new distributed/parallel services these kernels have to offer.

Claim 10 *Flexible/extensible kernel mechanisms for cluster-wide resource management help match the needs of the parallel and distributed applications run on*

top of COTS clusters.

The traditional kernels failed to adapt properly to the new emerging paradigms of computation. Especially the Internet and multimedia explosion showed that the conventional kernels cannot yield the best performance, as many of their inner policies mismatch the needs of the applications. Hence, the call for flexibility/extensibility in the kernel operation. This thesis pleads in favor of flexible kernel policies that enable the applications to take part in the resource management by specifying their expectations to the kernel, which, in turn, is supposed to honor them. For instance, we show that by taking into account information about Web request distribution curves, the kernel level cooperative caching performs better than its general purpose counterpart. Also, by downloading into the kernel locality-aware request routing policies that use the TCP connection endpoint migration and information gathered from the Web request distribution curve, it is possible to outperform classic request dispatching solutions such as Round Robin.

1.4 The road ahead

In the next chapter we present the software architecture of our SSI Web server based on a general purpose, policy-oriented request routing algorithm. Then, the rest of the thesis can be followed easily by looking at Figure 1.1, namely on the path leading from the lowest level (that of the communication subsystem) upwards to the user-space application server. In fact, only the kernel level subsystems will be of concern to this thesis. Chapter 3 presents the architecture of the CARD drivers, followed in Chapter 4 by the description of the cooperative caching policies (in particular, the description of Home-based Server-less Cooperative Caching) and the way they steer the CARD operation. One step further, Chapter 5 describes a SSI Web server using hints from the request distribution curve in order to improve the performance of HSCC. Chapter 6 presents the TCP connection endpoint migration protocol and the way such a service can be used in back-end level request distribution policies, as part of our general purpose three-phase request routing algorithm. The description of speculative TCP connection admission follows in Chapter 7. We summarize our results and present future work in Chapter 8.

Chapter 2

Server architecture

This chapter sketches the architecture design of our SSI cluster-based server. One of its main features is that it concentrates most of the functionality at the back-end level while keeping the front-ends involved as little as possible. This design choice is consistent with our endeavor to build cluster-wide mechanisms and policies that offer to the external user the image of a single system. Equally important in our design is a platform for cooperation among such cluster-wide SSI services. In the context of a Web server, we are interested to support the request dispatching with appropriate mechanisms that offer a node the possibility to switch strategies between data migration (through virtual disks and cooperative caching) and connection migration. The rest of the chapter revolves around these two issues.

2.1 Related work

Building locally-distributed servers is a topic to which a lot of research work has been devoted. Most of this work has been done for a particular type of service, the Web. For this reason, the rest of this section will discuss the previous work in terms of Web servers and will point out as needed the distinction to our approach.

A broad survey of the accomplishments in this field can be found in Cardellini et al. [18], which offers also a taxonomy that we will follow in this section as well. In fact, this section builds mostly on the information available in Cardellini et al. [18]. However, the survey does not discuss the TCP connection migration among the presented request routing mechanisms. We will present in Section 6.1 the research work related to the TCP connection migration.

One can classify the Web servers according to two main criteria: the request routing mechanism used and the server architecture. Depending on the request routing mechanism used, one can identify systems that do:

- *client Web routing* - assumes that the client itself is responsible for assigning requests to the servers.
- *DNS-based routing* - assumes that the requests are routed to the servers according to the hints provided by the authoritative DNS server for the targeted Web site.
- *network level routing* - delegates the request dispatching job to router devices
- *Web service level routing* - assumes that the Web server itself or custom dispatching devices placed in front of the Web server are dealing with the request assignment.

According to the type of server architecture employed, there are:

- *Cluster-based Web servers* - choose to hide the IP addresses of the back-end servers from the clients. Typically, the clients access the server through a public *Virtual IP (VIP) address* assigned to the front-end(s). Such a front-end, also called a *Web switch*, acts as a centralized dispatcher for the incoming requests.
- *Virtual (ONE-IP) Web servers* - use a public *Virtual IP address* as well, but this address is assigned to each of the back-end servers.
- *Distributed Web servers* - advertise their real IP addresses to the clients.

The last two types of architectures do not employ front-end dispatching. As expected, each of these architectures uses its own request dispatching scheme.

2.1.1 Request routing in cluster-based Web servers

The cluster-based Web servers use *TCP-layer* (*layer-4* according to the OSI standard) or *application-layer* (*layer-7* according to OSI) switches to assign requests to the back-ends. The *TCP-layer* switches implement *content-blind (uninformed) routing*, while the *application-layer* switches may use *content-aware (informed) routing*. A special discussion on *informed routing* is provided at the end of this section (Subsection 2.1.4).

The *TCP-layer* switches use techniques such as *packet rewriting*, *packet tunneling* or *packet forwarding* to route requests inside the cluster-based server. *Packet rewriting* changes the VIP address with the proper IP address of the chosen server. *Packet tunneling* encapsulates IP datagrams within IP datagrams addressed to the

servicing back-end. *Packet forwarding* uses MAC (Medium Access Control) addresses instead of IP addresses to identify the back-end. It is assumed that all the back-ends share the same VIP address and lie on the same physical LAN segment.

The *application-layer* switches perform an application protocol analysis in order to assign better the requests to the back-ends by using a content-aware distribution. They are slower than the *TCP-layer* switches, but offer better amortized performance. The actual request routing is achieved through *TCP gateways*, *TCP splicing* techniques or *TCP hand-off* protocols. The *TCP gateways* keep opened connections between the switch and the back-ends and forward the incoming requests along these connections. *TCP splicing* attempts to improve on *TCP gateways* by forwarding packets at the network layer between the network card and the TCP/IP stack in order to avoid the expenses of the application protocol processing. With *TCP hand-off*, the switch passes the connection endpoint to the back-end server which communicates directly with the client. In order to accommodate persistent HTTP connections, TCP hand-off has been further extended to *multiple hand-off* [7], which enables the front-end to successively pass a connection endpoint to multiple back-ends. *Back-end request forwarding* [7] improved on multiple hand-off by forwarding both the requests and the responses through the back-end currently managing the connection. Thus, the overhead of switching between back-ends is avoided.

In general, the *TCP layer* switches outperform the *application layer* ones. The request distribution is done earlier (at the connection setup time rather than once the connection exists) and the burden of the application protocol processing is being avoided. The switch itself raises the well known problems of being a performance bottleneck and a single point of failure.

2.1.2 Request routing in Virtual Web servers

The virtual Web servers must filter the incoming requests as these hit at the same time all the back-end machines sharing the VIP. The filtering process is distributed among the back-ends and must designate the machine that will service the requests. This filtering is typically a matter of computing a hash function on the source IP address and sometimes also on the source port number. If the hash value matches the server value, the request is accepted. The request distribution is uninformed (content-blind) because the back-end servers identify themselves by examining information at TCP/IP level (client IP and port number). The request routing is done at MAC level. As an important observation, routing requests at MAC level has the advantage of accommodating any IP-based service. It avoids also the single point of failure and the bottleneck problems. However, the main drawback stems from the inability to take advantage of informed routing.

2.1.3 Request routing in Distributed Web servers

The distributed Web servers make visible the IP addresses of all the back-end machines. Historically, the first distributed Web servers used round-robin DNS [16] to route requests to the actual servicing nodes by relying on address resolution to provide every new name query with another IP address. However, the intermediate address caching at the clients and the non-authoritative DNS servers causes severe load imbalances and this scheme ended up being used as part of two-stage request distribution algorithms. The second routing step takes place as a distributed algorithm at the Web server level. Such schemes include *third party relaying*, *HTTP redirection* and *URL rewriting*.

In *third party relaying*, a server is chosen using round-robin DNS. Depending on the local conditions, this server may choose to relay the requests to another server. All the incoming packets exchanged between the client and the new server will continue to flow through the first server. However, the new server responds directly to the client. The obvious disadvantage of this solution is the extra hop between the client and the server.

HTTP redirection is a standard protocol in which Web servers redirect client requests to other servers by using an HTTP status code. This technique has the advantage of handling the request routing at the Web page level, which obviously allows for informed request distribution. However, the system overhead of redirecting the request is considerable as it implies setting up a new connection, which can be quite expensive on slow networks and wastes both resources and processing time.

URL rewriting is another redirection mechanism which changes dynamically the hyperlinks within the requested page so that they point to another node. However, this dynamic generation of Web pages may prove expensive. Also, the new node name has to be resolved as well and that resolution incurs additional DNS queries. As with *HTTP redirection*, *URL rewriting* can be handled at user level and, for a given client-server pair, further communications can be redirected with a single request.

Our approach to the server architecture and request distribution is somewhat out of the above taxonomy. The closest design to our server is that of the *cluster-based servers*. The reasons for not matching any of the previously mentioned classes were manifold. First, we chose to design a system that capitalizes at maximum on the past experiences, and, naturally, that choice imposed a mixture of the techniques previously mentioned. Second, we want to improve on these solutions and therefore we devised new mechanisms and policies that found no appropriate design in the past attempts. And last, but not least, we are interested in developing general methods for building scalable servers, and some of the previous solutions

in the area of the Web server development were too particular to the Web service (see mechanisms such as *HTTP redirection* or *URL rewriting*).

2.1.4 Informed request distribution in cluster-based servers

Finally, we will discuss briefly some informed (content-aware) request distribution systems. As seen from the general taxonomy presented at the beginning of this subsection, only some of the server architectures and routing mechanisms are suited for informed request dispatching. In fact, most of the solutions so far were implemented for the class of servers designated as *Cluster-based Web servers*. Systems such as those described by Yang et al. [73] or Zhang et al. [75] are *application layer* switches and therefore have the aforementioned disadvantages.

One of the most interesting and complex solutions to the informed request distribution in cluster servers aims at reconciling the load balancing with the data reference locality at the back-end level. Such solutions are also known as *locality-aware request distribution*, shortly *LARD*, a term coined by the paper [52] which introduced the issue in the literature. The issue is critical, as the two goals are opposite: a perfect load balancing hurts the data locality, while striving to achieve good data locality breaks the load balancing. Finding a trade-off between the two is a difficult task. Some contributions in the field so far [52, 17, 2] chose simulations to validate their work. A LARD prototype [52] employed a TCP hand-off protocol. A follow-up paper by Aron et al. [8] made a point for moving the content-aware request distribution at the back-end level. Ahn et al. [2] discuss content-aware cooperative caching.

2.2 Design challenges

By looking at Figure 1.2, one can identify some architectural challenges. First off, the use of specialized front-ends for the request dispatching represents a centralized, single point of failure and a possible performance bottleneck. The typical solution to this problem is the front-end replication. We would like our system to provide such functionality.

Second, most of the time the back-end server machines act independently of each other and may thus cause a serious loss of cooperation potential as pointed out by the experience of the content-aware request distribution systems. In the case of the COTS clusters, the presence of the SANs advocates even further for a back-end cooperation, as such interconnects benefit of latency and bandwidth figures comparable to those of the memory subsystems.

Third, such cluster-based servers are application-insensitive. The server software runs on the back-ends, while the request routing takes place at the front-ends. Therefore, the request routing is unaware of the needs of the server applications. This shortcoming makes a case for developing back-end level request routing mechanisms (as advocated by Aron et al. [8] as well).

But this issue goes even further and raises server-specific problems. As we saw in the previous section, finding a trade-off between data locality and load balancing is a hard task and requires specific back-end level cooperation. In fact, the issue is complicated even further because the cluster-based servers are most of the time used in complex environments using proxies, content delivery networks, etc. The previous research [54] has shown that, in such setups, the request stream gets filtered out at early stages and only the requests for large, unpopular static documents or dynamic content reach the cluster-based server. This effect is known as the "trickle-down" effect. As a consequence of it, the pressure on the server storage system increases and the locality-aware request distribution becomes crucial. The side-effect of it is that the load balancing at the back-end level loses importance.

To summarize a bit, the lessons of the past experiences advocate for an extended cluster-wide cooperation at the back-end level, for single system image constructs that hide the individual back-end presentation and for the need for improved caching in the I/O subsystem of the server. But all these issues bring us back to the discussion about the need to have a cluster operating system providing the corresponding cluster-wide services. Our solution to the server-specific cluster-wide SSI services relies on extending the traditional stand-alone kernel software through "resource virtualization". The remote resources are virtualized locally over the SAN through specific constructs of the Virtual Resource Layer (see Figure 1.1). The two main services that we will further present, the cooperative caching and the TCP connection endpoint migration, are the main ingredients of a general purpose request dispatching algorithm representing the core of our cluster-based server. The next section presents this algorithm.

2.3 Server and request distribution architecture

The server architecture uses the basic setup presented in Figure 1.2. The cluster nodes represent the back-end server machines. On each of these nodes runs an instance of a stand-alone server program (in our experiments we use the Apache server program [5]). The System Area Network links all the cluster nodes and acts as a communicating backplane among them. Most of the intra-cluster specific protocols take place on this communication backplane, while the traditional communication to the "world" employs the Local Area Network (LAN) facilities. The

low latency and high bandwidth of the SANs represent the incentives for the design of the SSI services that we further describe. The communication infrastructure is described in Section 2.4.

The front-ends presented in Figure 1.2 may play an active role in request distribution or may be totally circumvented by the request dispatching schemes. This issue pertains to a more refined discussion, mostly related to the performance sensitivity. Since any intranet has a gateway to the Internet anyway, avoiding the involvement of the front-ends in the request distribution may seem overcautious as long as these front-ends deliver sufficient performance. However, as we will discuss in Chapter 6, a fully-distributed request routing solution avoiding the front-end(s) involvement may pay off. Moreover, such a solution would fit perfectly the SSI cluster-based servers that rely on back-end level mechanisms only.

Following the taxonomy presented in Section 2.1, our architecture design employs features also present in *Cluster-based* and *Virtual servers*. As in the case of the *Cluster-based servers*, we may employ front-end machines as *TCP-layer* switches. Nevertheless, the back-ends can also take over the TCP routing in a fully distributed design by using a connection endpoint migration protocol. Similar to the *Virtual servers* solution, the *Virtual IP* address of the server is assigned to each of the back-end machines and not to the front-end(s). However, unlike the *Virtual servers* solution, the purpose of the *Virtual IP* is different. The *Virtual IP* is not used to enable the incoming requests to hit simultaneously all of the back-ends because we find the request filtering procedures of the *Virtual servers* to be too costly. Instead, we use the *Virtual IP* to impose a Single System Image view on the cluster-based server and to establish a functional equivalence of all the back-end nodes in order to make the TCP connection endpoint migration effective. A (*Virtual IP, service port*) pair defines a *generic* TCP endpoint. The connections linked to this generic endpoint can migrate from one physical server to another through the connection endpoint migration algorithm we will present in Chapter 6. The reason behind this mixed approach is to restrict the functionality of the front-end to that of a connection router (if at all used), while all the main informed request distribution decisions take place at the back-end level.

2.3.1 Request distribution architecture

The central point of the software architecture of the SSI cluster-based server is a request distribution scheme expressed as a two-phase (three-phase, if front ends are used) algorithm. If front-ends are to be used, the first phase is executed at the front-ends, while the other two are taking place at the back-end level and express user-specified policies. Our main concern when designing the request distribution system was to keep it as general as possible, that is, independent of any particular

service. Also, we strove to identify the classes of mechanisms that can turn out to be helpful for such systems and can be implemented as operating system pluggable services (modules). Therefore, our design choice went more on the system side and explains, for instance, the use of the *TCP-layer* switches instead of the *application-layer* ones in our request routing algorithms. The attempt to present the cluster as a single system motivated the choice of assigning the *Virtual IP* to each back-end machine. This choice is consistent with our decision to handle the request distribution mainly at the back-end level, where each node should be able to identify itself not only as a stand-alone machine but also as part of a distributed server.

Nevertheless, our design does not understate the importance of the specific features of the application. However, instead of relying on application-level solutions only, we aim at a deeper collaboration between the applications and the underlying kernel. Capitalizing on the extensible/grafting-capable kernels experience [14, 28], our design allows the applications (in this particular case, the server program) specifying their own policies and downloading them into the kernel as phases in the request distribution algorithm. Therefore, our request distribution is *policy*- and not *service-oriented*. Most of the time, these policies are supposed to perform not only the application specific processing but also to improve the overall performance of the cluster by taking into account the side-effects of the service protocol processing. But for a better understanding of how the policies work, we continue by presenting the request distribution algorithm of our system.

When used, the front-ends perform a blind (non-informed) request distribution. In fact, it is a simple connection routing based on a hashing function that attempts to distribute uniformly the requests. The distribution is deliberately simple in order to offload the front-end machine. There can be many such connection routers, which is an important scalability factor. Each generic endpoint of a connection at the back-end level "remembers" the front-end router through which the request "came".

The first back-end level distribution phase takes place when the incoming connection requests targeting the generic TCP endpoint hit the back-end nodes. We call it the *unconnected* phase, because it occurs early, at the connection setup time. This phase is entirely devoted to the load balancing of the cluster nodes. Depending on the policy used, an incoming SYN packet may trigger a check on the TCP and CPU load of that machine. If the machine is heavily loaded, the incoming SYN packet is redirected to a less loaded machine. Finding such machines depends also on policies. For instance, in Chapter 4 we will see how HSCC, our cooperative caching algorithm, identifies heavily loaded nodes based on the number of the disk blocks locally cached on behalf of other cluster nodes. Anyway, the price paid for this extra hop is the SAN latency for a SYN packet, amortized over the cost

of the entire connection activity. Since this SYN redirection happens only once at the connection setup time, its price can be considered negligible. For the servers that use front-ends, handling the TCP connection endpoint assignment at the back-end level is consistent with the decision to keep the front-end machines as lightly loaded as possible, since their task is mainly to interface the back-end machines with the "outside" world. Moreover, it is a natural choice as the back-end machines are supposed to cooperate and, therefore, to have the kind of information that enables them to choose a better servicing node for the request. Otherwise, one would need to inform the front-end node about the loads of the various back-end nodes and to involve it in a more complicated request dispatching algorithm.

The second back-end level phase of the request distribution concerns the back-end machines already connected to clients, either as a result of a front-end or a back-end redirection. This is the *connected* phase, it is entirely policy-driven and may result in a TCP connection endpoint migration. The normal application-level protocol (HTTP, for instance) processing is carried out but it may be influenced by hints provided by the distribution policy. In the case of a LARD policy, for example, when a file is to be opened for access, a decision has to be made as to whether the request will be serviced locally or it will be relocated it to another node. Such content-aware decisions would be mainly heuristic methods of choosing between data and connection migration.

A special type of a connected back-end level policy helps perform informed load balancing entirely at the back-end level. We called it speculative TCP connection admission. The general idea is that the front-ends shouldn't pry into the incoming requests at all. Instead, the requests hit a back-end server and only there, based on cluster-wide knowledge, a policy that uses the TCP connection endpoint migration mechanism can balance the loads of the cluster.

2.4 The communication infrastructure

As already mentioned, the SAN infrastructure is quintessential to the needs of the SSI services that build upon it. As seen in Figure 1.1, the communication software layer is the basis of all the other cluster OS layers. Therefore, before proceeding any further, we take the opportunity to present its features and the services it offers.

One important aspect of the communication layer is its hardware independence. Namely, it defines an interface for the upper cluster OS layers which will use it to access the network hardware. The main services of this interface allow defining a cluster-wide unique node ID, message types, mapping/unmapping node IDs to/from network hardware addresses and, of course, sending/receiving messages. For every network hardware technology there will be a kernel module implemen-

tation of this interface that will be hooked up with the kernel. Throughout our experiments we used Myrinet cards under the control of the GM driver [66].

Defining a node ID abstraction is of capital importance to our SSI software as both the CARDs (and cooperative caching) and the TCP connection endpoint migration use this kind of information to identify nodes in the cluster. As we will see at the right time, the ID of the node exporting a disk through CARD drivers is part of the CARD driver identification mechanism together with the traditional major and minor numbers [9]. The TCP connection endpoint migration protocols use also the node IDs to designate both the source and the destination of a migration. The main routines dealing with IDs are:

- **ps_host_id** - returns the local host ID
- **ps_id_to_addr** - maps a given ID to a network hardware address
- **ps_addr_to_id** - converts a given network hardware address to a node ID

When using GM over Myrinet, the task of assigning unique IDs cluster-wide is directly supported by the GM software. Therefore, in this case, the node IDs are in fact the Myrinet IDs. The GM software supports also the conversion routines between Myrinet IDs and Myrinet hardware addresses.

The interface is also responsible for defining various message types for the use of the cluster OS. The block request messages issued by a CARD driver as a consequence of a local cache miss are labeled by a corresponding type, while a TCP SYN packet initiating a connection endpoint migration will receive its own type. Sending messages is accomplished by means of an **ps_send_msg** routine which takes as parameters a message type, a chunk of opaque data, the size of the message and, naturally, its destination address:

```
int ps_send_msg(struct device *netdev,
                unsigned short msg_type,
                unsigned short msg_size,
                void *msg,
                unsigned char *dst_addr);
```

Receiving messages follows an event-driven model. The upper layer kernel software “downloads” a specific handler for each type of message into the communication layer. As soon as a message of a given type arrives at the node and the network interrupt routine delivers it to the host, the corresponding handler fires up and performs the necessary message processing. This handler runs in interrupt context as a software interrupt handler (a.k.a. *bottom half* in Linux). The idea of

associating messages with the code needed to process them is close to that of Active Messages [69], although in our case the message doesn't carry the code within itself, as such a solution would have been impractical given the size and complexity that such handlers might have.

In order to make this infrastructure truly useful, we will describe in the next chapters various policies and the way they interact with the request distribution algorithm. We support the data migration by using CARDs (Cluster Aware Remote Disks) and cooperative caching, while the request migration is accomplished by means of the TCP connection endpoint migration.

2.5 Summary

In this chapter we described the general architecture of our cluster-based server and its request distribution algorithm. The server employs a unique Virtual IP associated with each of the back-end machines of the cluster. The request routing may or may not involve the front-end machines. The global request dispatching algorithm relies on a policy-oriented design that ensures independence of the application specificity. The applications are permitted to download their instructions into the kernel by means of policies that operate as phases of the global request dispatching algorithm of our cluster-based server.

At the end of the chapter we also presented briefly the architecture of our hardware-independent communication layer. This layer defines the node IDs that are used by all the other upper OS layers, as well as the conversion routines that allow mapping these IDs to hardware addresses and vice-versa. The send/receive messaging system is typed. Receiving messages is event-driven, a special handler being called to process an incoming message according to its type.

Chapter 3

Remote I/O

The tremendous potential of the COTS clusters cannot be always fully exploited as the building block, the PC system, suffers from various well-known problems. One widely recognized problem is the disk I/O bottleneck [45]. The ever increasing gap between the rates at which the processor and disk speeds grow, the additional pressure put on the disk I/O subsystem by the increased data sets, or the lack of cooperation among the optimization algorithms used by the various system caches (at the OS level or inside the disk controller itself) represent some of the most challenging issues related to this problem.

The clusters suffer naturally from these problems but their distributed architecture raises additional concerns. First off, a software architecture like the one depicted in Figure 1.1, which relies on a Virtual Resource Layer, has to provide proper abstractions to the upper kernel layers. In the case of the disk I/O subsystems, these abstractions are traditionally the disk blocks (for raw access or databases that decide to implement their own data management mechanisms and policies) and/or the I/O pages (the file systems being the typical "consumer" of this type of abstraction). A typical answer to the question is represented by the *virtual disks* [46], that is, kernel drivers capable to fulfill local data requests by fetching remote data over the net (in our case, the SAN). Thus, our first endeavor is to design and implement an efficient kernel level virtual disk driver. The virtual disks help us share the data across the cluster without having to replicate it locally on each cluster node.

However, having a solution for cluster-wide data sharing is only the starting point of our concerns. Recent research on exclusive caching [72, 19] showed the importance of the cooperation among various system caches (the file system cache, the disk controller cache, etc.) in a single machine in order to maintain the control over multiple cached copies of the same data block. In the case of a cluster using virtual disks, the cache hierarchy grows beyond the boundaries of the local sys-

tem and the local operation can affect implicitly the operation of the remote node accessed through the virtual disk. To complicate the matter even further, if one considers the problem of the constantly growing speed difference between processors and disks, the natural answer is to enlarge the cache in order to accommodate larger working sets and, thus, to hope for improved performance. The good news about clusters is that they can rely on cooperative caching to take advantage of a global, cooperative, cluster-wide cache. Nevertheless, building and managing a global cache remains a tricky business. Depending on the complexity of the cooperative cache design, the capability of controlling which copies of a certain block should be kept around (potentially by using remote memories) pushes the issue of exclusive caching even further. In this chapter, we present a low-level solution for virtualizing computer-attached remote disks in COTS clusters. Following the upward path through the software architecture graph in Figure 1.1, the next chapter (Chapter 4) deals with our approach to cooperative caching.

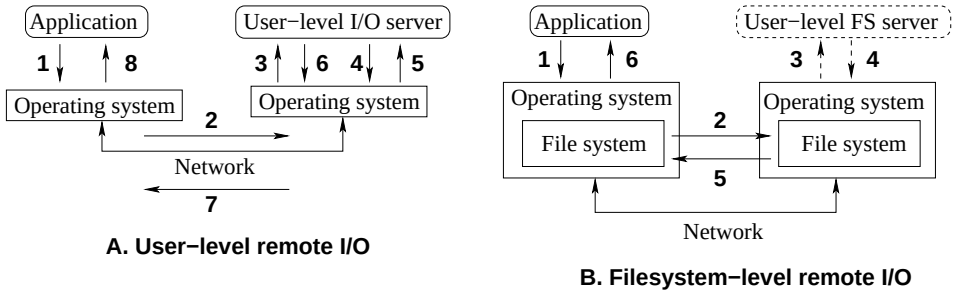
Our virtual disks are called Cluster-Aware Remote Disks (CARD) and virtualize the accesses to the remote computer-attached storage. The block requests sent to the remote disk nodes may bypass the remote disk caches by using a single copy protocol. A remote disk node using this protocol can send the disk data through DMA directly into the network buffers without storing a copy of it in the disk cache. The protocol provides exclusive caching by avoiding two copies of the same block, one stored in the remote cache and the other one in the local cache. The remote disk caches using exclusive caching are not affected by the incoming block requests. Thus, they are isolated from the external influence and, in such a case, a computer-attached remote disk behaves like a network-attached one. The remote disks are mounted locally by means of virtual disk drivers as any locally-attached disk would be and the file systems consider them local storage. Therefore, the driver performance must be comparable to that of the local drives. Our CARD drivers achieve the desired performance by using a highly asynchronous mode of operation based on an event-driven model of computation that attempts to make the most out of the SAN performance. Also, the influence of the semantics and the assumptions of the file system using the CARD driver is confined locally, so that they don't have adverse effects on the operation of the remote disk. In this context, we discuss the impact of the file system read-ahead policy. A CARD prototype implemented in Linux showed performance comparable to that of the local disks.

Parts of the material presented in this chapter appeared in [49, 48].

3.1 Related work

The solutions to improve the disk I/O concern various levels of single systems, as well as the cooperation among systems in the case of the remote disk I/O for paral-

lel/distributed computing. The remote disk I/O solutions can be broadly classified in: user-level-, file system- or low-level-oriented solutions. Figure 3.1 depicts the three possible approaches to the remote disk I/O. The arrows on the picture represent the individual steps of a remote disk I/O operation, while the arrow numbers show the order of these steps.



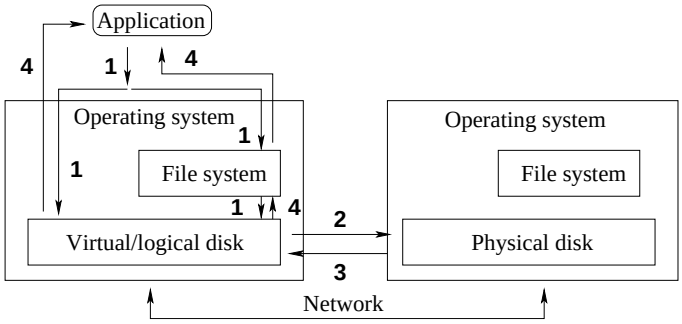


Figure 3.1: Remote I/O systems

In the user-level solution, an application program asking for a block contacts a remote user-level I/O server that, upon receiving the request (Figure 3.1 A, step 3), asks the local operating system to provide the block (step 4). As soon as the block is delivered to the I/O server (step 5), the server can send the block back to the requester over the network (steps 6 and 7).

The file system-level solution enables the applications to access remote data through the services of distributed/parallel file systems. Most of the time, such systems avoid handling requests in user-space. However, some implementations rely on an user-level request service (depicted in Figure 3.1 B by the dashed arrows, steps 3 and 4, and the user-level file system server).

The low-level remote disk I/O systems rely on the virtualization of the remote storage (see Figure 3.1 C). Inside the operating system, a disk I/O request issued by an application is handed over (either directly or through the local file system) to

a virtual (or logical) disk driver that is supposed to serve the request. The virtual disk driver fetches the blocks from the remote disk over the network.

The low-level stand-alone I/O systems improve the performance at the disk level either by bridging the speed gap between the processor and the disk (Active Disks [1]) or by integrating the various system caches into a cooperative I/O infrastructure. Exclusive caching [72, 19] avoids the double buffering occurring in independently managed caches (e.g., the file system and the disk controller caches). The file system-aware disk controller prefetching [57, 19] compensates for the mismatch between the read-ahead policies of the file system and the disk controller.

The distributed low-level I/O systems include Petal *distributed virtual disks* [46], *single-image I/O systems* (SIOS) [36], *striped log-based storage* (Swarm [35]) and *network-attached secure disks* (NASD) [32]. Petal is a block-oriented storage system that manages a pool of distributed physical disks by providing an abstraction called *virtual disk*. A virtual disk is globally accessible and offers a consistent view to all its clients. The SIOS Virtual Device Drivers (VDD) encompass the entire disk capacity of a node (both local and virtual disks) much like in a RAID system (actually implemented as default technology). Swarm offers the storage abstraction of a *striped log* while the NASDs provide an object-oriented interface.

At higher levels, distributed file systems like Frangipani [67] use Petal virtual disks and supply them with meta-data consistency support by implementing a distributed lock service. Server-less file systems [4] distribute the meta-data management and use cooperative caching [23, 55] to scale their working set beyond the limit of the locally available memory. The usual distributed file system memory hierarchy (local cache, server cache, server disk) is extended by adding the client caches, provided that reading from remote caches takes less time than accessing the disk. The PACA [21] parallel file system mixes cooperative caching with global memory and memory-to-memory copies (a form of Remote DMA or RDMA).

Direct access file systems such as DAFS [25] modify the distributed I/O memory hierarchy as well, not by adding but by removing a level, namely the local kernel cache. DAFS is based on NFS and addresses to a class of applications that have seldom sharing access patterns. DAFS runs in user space and uses RDMA to communicate directly with the file server in order to bypass the local operating system. Unlike PACA, no global memory support is provided.

Parallel file systems like Clusterfile [40] aim to match efficiently the access patterns of the parallel applications to the physical partitioning of the parallel files across the cluster. Studies [58, 47, 56] have shown that the mismatches between the two cause most of the performance loss in parallel file systems. Clusterfile minimizes such mismatches by allowing a flexible physical partitioning of the file across the cluster disks. By combining collective I/O [26, 44] with cooperative

caching, Clusterfile provides cooperatively cached collective I/O buffers [39].

Other user-level systems are mostly a work-around: block devices [43] using VIA [6] or file systems (PVFS [38]) in user space, remote I/O libraries [31] over MPI-IO [20]. Since the traditional kernels are unaware of the distributed nature of these systems and their inner mechanisms and policies fail to match the expectations of the user space driven computation, the end result is performance penalty. Also, moving typical kernel code in user space incurs increased application software complexity and more difficult development.

3.2 Background on disk drivers and the file system

The CARD drivers build a block-level distributed storage system and can support both file and database systems on top of them. However, all our work on remote I/O refers exclusively to the interaction of the CARD drivers with the file systems. In order to explain properly this interaction, we start out by presenting the operating system concepts needed to understand the operation of our CARD-based storage system. Presenting such concepts includes describing the basics of the disk driver operation and its interaction with the file system.

3.2.1 Disk drivers and the file system

The interaction between a file system and a disk driver is intermediated by a page or buffer cache that attempts to speed up the disk access. Usually, the file data is accessed through the page cache, while raw and special data (meta-data such as inodes and superblocks [9]) accesses use the buffer cache. Unlike the buffer cache, which is indexed by *disk ID* and *block number*, the page cache is indexed by *inode* and *offset* within the file. This particular indexing of the page cache allows an efficient implementation of the memory-mapped files (see the Unix *mmap* system call). Thus, pages store disk blocks contiguous in the logical file layout (a byte stream for the Unix files) but potentially uncontiguous on disk.

Regardless of the type of the requested data, *buffer_head* structures [9] are used to describe the in-memory copies of a disk block. These buffer heads are always the link between the corresponding disk cache and the strategy routine of the disk driver. This routine is responsible for passing disk jobs to the driver and possibly to schedule them for optimal access. In Linux, optimizing the disk access is done by coalescing disk requests for consecutive blocks, if possible. The disk request structures keep pointers to the buffer head(s) to which they correspond and use these structures to cooperate with the disk cache (either the buffer or the page cache). In the case of the buffer cache, the buffer heads are stored in a hash list and are called *synchronous* buffer heads as they can be later used to access the cached

blocks. For the page cache, only temporary buffer heads are used. As previously mentioned, they are used only to make the connection between the disk strategy routine and the page cache. Therefore, these buffer heads are called *asynchronous* (or I/O) buffer heads. While the synchronous buffer heads point to the buffer cache, the asynchronous buffers heads are used only to load blocks potentially uncontiguous on disk in a page that reflects logical contiguity, as instructed by the file system layout. Figure 3.2 shows the relationship of the asynchronous buffer heads to the I/O pages.

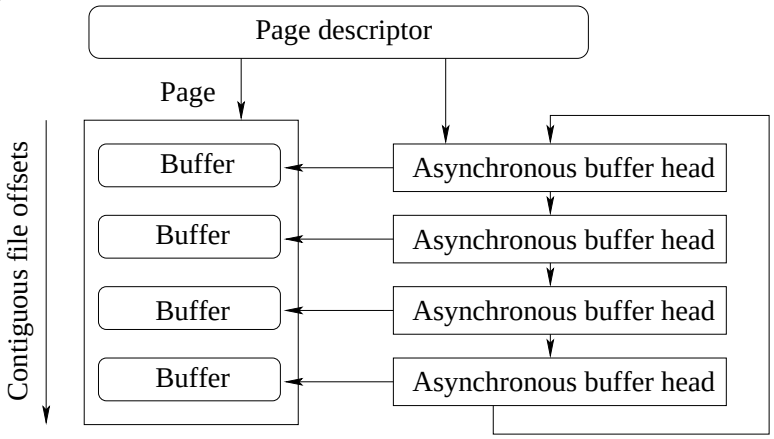


Figure 3.2: Asynchronous buffer heads

When reading from a file, a process issues a system call that uses the appropriate kernel read routine provided by the file system to deliver the requested data. The file system read routine checks the page cache for a page containing the requested data and, if found, delivers the data back to the user level process.

However, if the requested data is not cached, a new page is allocated and an appropriate asynchronous buffer head (or more if the page size is larger than the block size) is created to describe the in-memory copy of the disk block. The page is locked and a disk request gathering references to the asynchronous buffer heads is passed to the disk strategy routine. The process making the request goes to sleep, while the driver is responsible for serving the request.

When the driver finishes the service, a disk interrupt is issued and the corresponding interrupt handler is called. This handler is sometimes called the hardware interrupt handler (or the *upper half*). It is supposed to perform only the critical operations needed to handle the interrupt, while leaving the uncritical tasks for later handling through a so called software interrupt handler (or *bottom half*). Before completion, the upper half checks whether the situation of the system permits the execution of the bottom half as well. If so, the software handler is called and runs to completion (can be interrupted though by further incoming hardware interrupts,

unless it explicitly disables them; in any case, a bottom half cannot be interrupted by any other bottom halfs). Otherwise, its execution is deferred to a later time (more precisely, before the scheduler chooses the next process to run).

The disk bottom half dequeues the request, runs a callback routine associated with the asynchronous buffer head representing the block and wakes up any process that might wait for that block to become available. The role of the callback routine is to perform tasks related to the buffer head management. Typically, such tasks include marking the buffer head up-to-date and unlocking the page (or buffer, for synchronous buffer heads). Additionally, for asynchronous buffer heads, the callback releases the buffer head(s) as soon as the entire page has been validated. Reading raw or meta-data is similar, the only differences being that buffers and not pages are handled and that synchronous buffer heads are used.

The traditional path has been recently put under scrutiny especially by distributed file systems like the Direct-Access File System (DAFS) [25] that bypasses the local kernel in order to make a Remote DMA (a memory-to-memory access) to the file server. The disk blocks are fetched directly in the user-space and leave no trace in the kernel. Not storing a copy in the kernel cache assumes however a particular type of operation in which the block sharing on the client side is seldom. If this is not the case, mechanisms similar to those in the kernel are requested (a form of *mmap* with MAP_SHARED and Copy-On-Write).

Our approach is orthogonal to that of DAFS because we target different architectures. As mentioned, the clusters of PCs are supposed to represent highly-parallel machines, and the client-server assumptions of DAFS may not always fit the cluster paradigm. Therefore, we chose to use local stand-ins for the remote disks and to use them as if the resources were local. This decision raises questions at the file system level (for instance, questions related to the meta-data consistency) if stand-alone file systems are used on top of virtual disk drivers. However, we believe that it is possible to implement with minimal effort parallel/distributed file systems on top of the existing stand-alone ones, provided that the appropriate functionality is offered at the virtual disk driver level. The issues concerning such functionalities are presented in the next sections. However, the locking mechanisms and the meta-data consistency will not be discussed because these issues should be solved at the file system level and this topic is beyond the purposes of this thesis. For all the purposes of this thesis, a read-only storage system assumption suffices.

3.3 Design considerations

The CARD driver is a regular block device driver in the kernel. However, it virtualizes accesses to remote disks by fetching disk blocks over a SAN and that

implies meeting certain design decisions. First of all, one needs to establish an unique identity of a CARD throughout the cluster. Second, the existence of two disk caches in the kernel and the choice of a high-level abstraction cause problems as a CARD driver asks for a remote block. Since the CARD drivers operate on a block abstraction, they send along with the request a remote disk ID and a block number. However, at the remote site this pair cannot be used to index the page cache. Therefore, if the remote page cache has a cached copy of the requested block, the request goes unnecessarily to the disk. This scenario motivates the need for a dual behavior of the CARD driver: it should accommodate both I/O page and block requests.

Furthermore, having to deal with two systems, the CARD design has to decide to which extent the local requests will affect the remote page/buffer cache at the physical disk node, as the recent research in exclusive caching [72, 19] suggests possible benefits for certain classes of applications. Moreover, the separation of the various buffering systems inside the kernel incurs performance penalties. For instance, the network subsystem uses specialized socket buffers to send/receive messages, while the file system buffer cache uses its own buffering system (the buffer head cache). This separation implies that the CARD driver must perform two extra copies to move the data blocks between the two buffering systems. To avoid these copies, one needs a Remote DMA (RDMA) engine that allows a direct read/write access to remote memories through the SAN. Alternatively, one could use a unified network and cache buffering system such as IO-Lite [53]. We describe a partial solution consisting of a single copy protocol that allows also a simple exclusive caching implementation.

Additional concerns regard the local file system level policies like reading ahead. They may lose their efficiency if not properly exported to the remote system. And last but not least, the driver should make the most out of the available potential for parallelism in order to represent a true alternative to local drives performance-wise. In the next sections we will address each of these issues.

3.4 The software architecture of the CARD drivers

Figure 3.3 depicts the software architecture of the CARD storage. It comprises two main parts: the CARD kernel driver used to mount locally a remote disk and a protocol handler running at the remote site in order to fulfill the remote disk requests.

When a block request on a node misses in the local cache (step 1 on Figure 3.3), a request for that block is queued in the CARD driver while this one attempts to fetch the block from the remote disk (step 2). To do that, the CARD driver sends a request message (step 3) to the remote node and waits for the response.

The message passes over the SAN (step 4) and, once arrived at destination, it is delivered to the protocol handler (step 5). The handler represents code running as a software interrupt and is responsible to handle the incoming requests. First, it checks the local cache for a cached copy of the requested block (step 6). This step may be bypassed, as we will see later, by directly accessing the disk. If a cached copy of the block exists, it is delivered to the requester by sending a message back over the SAN (step 9). Otherwise, a request is addressed to the disk (step 7), which in turn fulfills it by filling out the local cache with a copy of the block (step 8). As soon as the copy becomes available, the data is sent back to the requester (step 9). The response travels back to the requesting node (step 10) where the CARD driver is being notified of the block arrival and fills up the appropriate cache block (step 11). Then, the process that requested the block is being awaken and given the block (step 12).

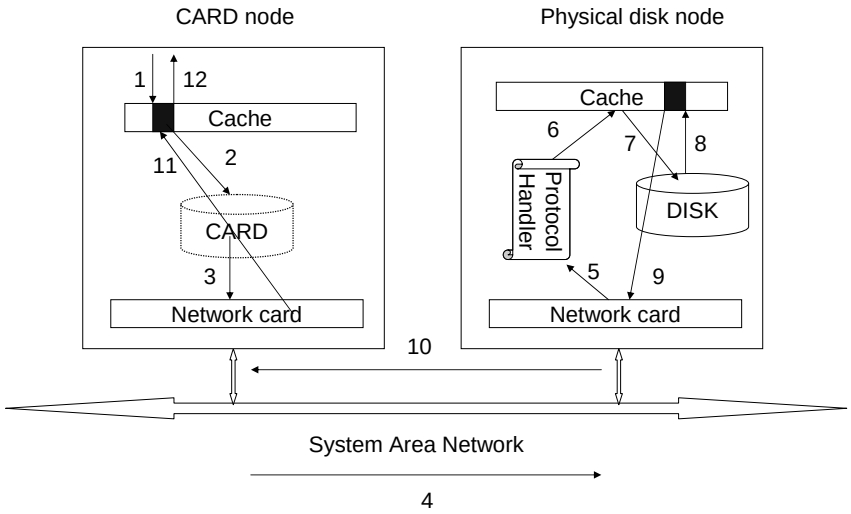


Figure 3.3: The software architecture of the CARD drivers. The numbers on the picture mark the steps taken by a request for a block miss in the local cache

3.5 Device identification and block addressing

In order to send a block request to a remote disk, a CARD driver has to be able to identify that node. Both the physical disk and the CARD driver virtualizing it are uniquely identified in the cluster by a tuple *(major number, minor number, physical id)*. The *major* and *minor* numbers form the conventional pair used to

identify a block device in stand-alone kernels. Many CARD drivers mounting the same remote disk on different nodes may have different *major* and *minor* numbers but share the same *physical id*. This ID is the ID of the physical disk node and, at the same time, the ID provided by the communication infrastructure (i.e., the result of calling *ps_host_id* as seen in Section 2.4). The major and minor numbers are used in the traditional manner to distinguish among the various local block drivers, while the third key of the tuple, the *physical id*, helps discriminate against different CARD drivers with the same *(major, minor)* pair. The locally cached blocks are found using the native addressing scheme of the local kernel based on major, minor and block numbers. The remotely cached blocks are first checked against their *physical id* to identify the actual CARD driver on that machine. Then, the native addressing scheme of the local kernel is used. Of course, this global addressing scheme assumes that a node mounts locally only once a remote disk by means of a CARD driver.

3.6 Dual operation: between pages and blocks

For a cluster-wide file system, a virtual disk driver offering only block-oriented access would complicate things. For one thing, systems implementing cooperative or global disk caches [4, 21] will not be able to optimize the memory usage. Imagine a node asking for a file page not locally cached. The request gets translated into a disk block request by the virtual disk driver and sent to the remote disk. The remote disk will receive a request containing a disk ID and a block number. If the remote file system already accessed that block not long ago, a copy of it should be in the page cache. But the page cache is indexed by inode and file offset, and thus the request will unnecessarily have to go to the disk. The scenario is equally valid for the cooperative caching case. Even if there is no available copy whatsoever, possible future requests to the same block at the remote node may benefit of a cached copy. Again, since only the disk ID and the block number were provided in the request, a copy at the remote node can be saved only in the buffer cache. Subsequent file accesses to the block will copy the content also in the page cache and thus will result in poor memory management (double buffering).

In order to answer the problem, one has to investigate two possible solutions. The simplest would be a page-based interface to the storage. Since the file systems define their logical layout most of the time as a sequential stream of bytes (model inherited from the Unix file systems), a page-based interface to the storage would make sense because it fits well with the memory-mapped file paradigm. Two arguments speak against this solution. First, our CARD-based storage system is not intended for file system exclusive use. One can easily build database systems on top of the CARD drivers. However, most of the time, the databases

avoid relying on file system facilities to manage their data. Therefore, offering just a page-based interface may not properly suit the needs of such systems. Second, the network-attached secure disks [32] propose an object-oriented interface to the storage system, as opposed to the traditional block-based one. The CARD drivers could support such an approach by adding another level of indirection between an object-oriented interface presumably offered by an object-oriented file system or database and the block-based interface. Having only the page-based interface would add an undesired level of indirection between the two, and thus would cause an additional overhead.

As a result of these considerations, we decided to provide our CARD-based storage system both with a page- and a block-oriented interface. This design choice admits to two implementations. One of them considers unifying the page and the buffer cache interfaces in the kernel, while the other one relies on extending the CARD functionality by sending page cache specific information along with the block requests. Both methods are described and discussed in the following subsections and both have been implemented in the Linux kernel.

3.6.1 The unified page-buffer cache

One possible solution to the problem is to re-engineer the kernel page cache management algorithm. The main idea is to keep around synchronous buffer heads that allow indexing the page cache through (disk ID, block number) pairs as well. Keeping around synchronous buffer heads is accomplished by modifying the low-level routine that starts page I/O operations (*brw_page* in Linux 2.2) as follows. As seen before, when filling out an I/O page, the system creates asynchronous buffer heads that point into the I/O page to help direct the DMA disk transfers ordered by the disk driver. Using synchronous buffer heads instead of asynchronous ones and inserting them into the buffer cache queues as well causes subsequent buffer cache lookups to succeed in finding the synchronous buffer heads that point to data actually stored in the page cache.

The solution seems simple, but there is a catch to it. Namely, when the free system memory becomes scarce, the memory management subsystem uses a clock algorithm [65, 9] to page out some of the memory pages. If the decision targets an I/O page, the page is saved to the disk only if it is modified (because an unmodified copy of it exists already in the corresponding file somewhere on disk). Nevertheless, the eviction of the page entails releasing the corresponding structures that index the page cache. At this point, the pages that are doubly indexed, both through the page and the buffer cache, should also get rid of the corresponding synchronous buffer heads that refer to the page. These additional deallocation operations imply kernel code changes in the clock algorithm (expressed by the *shrink_mmap* routine

in Linux 2.2).

The unified page-buffer cache approach is problematic, as in general kernel changes are hardly accepted by the users community, unless the problems they respond to cannot be solved otherwise. Moreover, it defies our attempt to keep most of the kernel unaware of the parallel/distributed environment in which it operates. Conceptually, the page cache needs no knowledge about the origin of the cached data. This kind of information may be kept in the file system, but belongs naturally to the storage subsystem. This affirmation holds for stand-alone systems and we aim to show that it extends naturally cluster-wide. Other obvious disadvantages of the hard-coded solution are its increased complexity and the additional bookkeeping memory consumption as the page cache keeps around synchronous buffer heads. The obvious advantage of this solution is the reduced CARD driver complexity, as the driver has to offer only the block-oriented interface.

3.6.2 The dual-behavior CARD driver

A cleaner solution to the problem doesn't affect the kernel but complicates the CARD driver design. Moreover, the CARD drivers stop being block-oriented storage and borrow file system specificity. The main advantage is the flexibility. At the storage level, it is possible to decide which copy to maintain: that in the page cache, that in the buffer cache or, as we will see in the following section, no copy at all.

When inserting disk requests in the CARD driver queue, the strategy routine decides whether the requested block should be looked up at the remote disk site in the page cache or in the buffer cache by setting an appropriate flag in the remote block request. If a page cache copy is desired, the inode and the file offset corresponding to the requested disk block are sent along with the request over the SAN. Although at this level only the block number and the disk ID should suffice, some kernel data structures make it easy to find this kind of information, namely by indexing in a page table (the *mem_map* array of *page* structures in Linux 2.2) holding pointers to the structures describing the usable memory pages. Since the disk data is stored in I/O pages, the *page* structure describing the page holds also pointers to the inode of the corresponding file and the offset within it.

At the remote disk, the protocol handler (see Figure 3.3) honors the flag and translates the received (inode, offset) pair into a locally valid index pair. This translation is needed because Linux doesn't use inode numbers for indexing, but the kernel addresses of the corresponding inode structures. This way, carefully coded hash schemes exploiting the placement of the inode cache in the kernel address space improve the lookup performance. Following the translation, a local access to the page cache is simulated as if it were issued by the local file system itself.

This procedure starts by looking up the page cache for a cached copy of the data. If not found, a new page is allocated and handed over to the normal I/O processing described at the beginning of this chapter. As a side effect, one gets also the bonus of prefetching the corresponding inode in the inode cache. This prefetching speeds up future (both local and remote) accesses to the same file.

3.7 Single copy protocol

The separation of the various buffering systems in the kernel affects the design of the virtual disk driver. The network interface uses specialized socket buffers to send/receive messages, while the file system uses its own buffering capabilities (the page/buffer cache). Separate buffering systems imply additional copying both at the local and at the remote disk nodes.

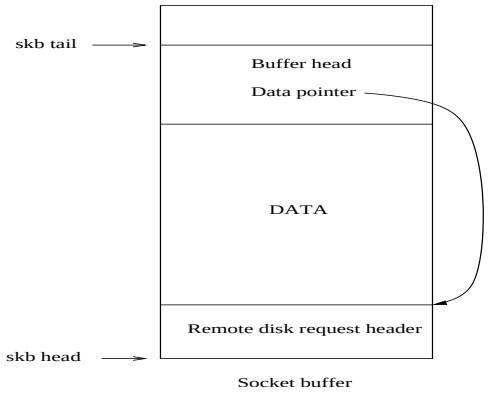


Figure 3.4: Socket buffer holding both block data and the associated buffer_head

Let's take the example of a file read operation. The virtual disk driver prepares locally a page to store a block and sends the block request over the SAN to the remote disk node. There, if already cached, the block is copied to the socket buffer and sent back to the requester. This copy could be avoided only if the SAN card was capable to perform a DMA from the page/buffer cache to the network card. Current RDMA implementations [25] for SANs do not allow such things since they require pre-defined pinned-down buffers. However, theoretically, the use of arbitrary buffers should be possible, as the disk drivers, for instance, have no problem sending their data through DMA to randomly chosen (i.e., not pinned-down) addresses like those of the individual pages/buffers.

For the uncached blocks we can do better. We instruct the disk at the remote node to perform a DMA into the response socket buffer instead of the page/buffer cache by preparing a socket buffer large enough to hold not only the data block, but

also the buffer head structure that describes its in-memory copy (see Figure 3.4). Thus, we bypass both the buffer head cache and an additional memory allocation for the data itself. The latter is possible as we use pre-allocated response socket buffers. We fill in the buffer head allocated in the socket buffer with the appropriate values and pass it to the strategy routine of the disk driver. As a result, the destination address for the disk DMA transfer registered in the disk request will be that provided by the buffer head which, in turn, points to an address within the response socket buffer itself. When the disk read operation completes, the disk software interrupt marks the "buffer" up-to-date and "releases" the buffer head. In fact, nothing gets done since the buffer head occupies memory in the socket buffer past the useful region and that memory won't be transfered back over the network.

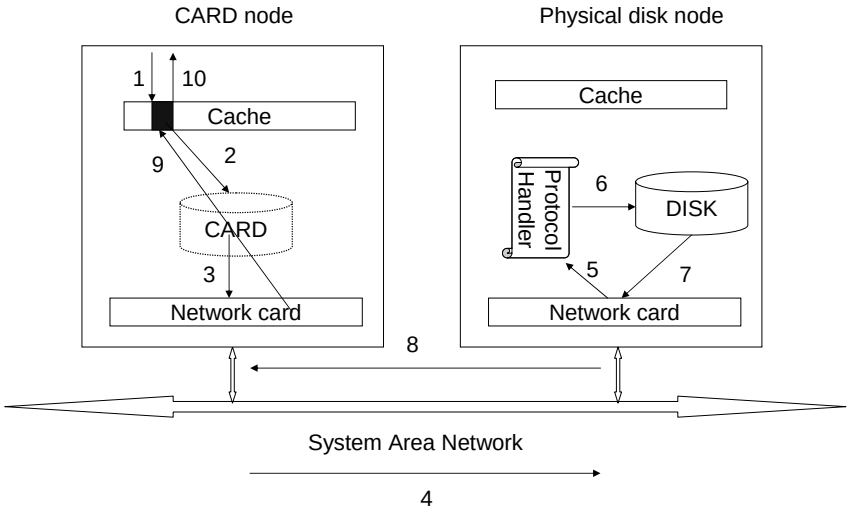


Figure 3.5: The exclusive caching operation of the CARD driver

As soon as the response arrives at the initiator, the virtual disk driver that issued the request has to copy the block data from the socket buffer to the page cache. Again, this copy could be avoided if the SAN card had a DMA capability enabling to send the data directly to the page cache. Or, as a general alternative to any DMA capability, one could use a unified network and cache buffering system such as IO-Lite [53]. The current CARD prototype doesn't support such techniques. A visual description of the whole protocol is given in Figure 3.5, which should be compared to the standard operation depicted in Figure 3.3.

The capability to control whether copies of the fetched blocks are left behind in the page/buffer cache at the remote disk node offers extended flexibility. On

one hand, a fine-grain exclusive caching [72, 19] is possible by simply sending an exclusive caching remote disk request (in fact, it is a matter of setting a certain flag in the request). On the other hand, the ability to control the copies at the remote site enables a dual behavior for the disk: either computer- or network-attached. When the node hosting the physical disk is not logically involved in the computation, the single copy protocol makes the remote disk look like network-attached storage.

3.8 The impact of the file system read-ahead policy and the disk fragmentation

The traditional stand-alone kernels use a file system read-ahead policy to improve the disk usage and throughput. The default action is to sequentially prefetch a given number of blocks. The policy adapts its behavior to the file access pattern of an application by using a read-ahead window that may shrink to zero when the accesses become random. This policy is complemented by the optimizations performed by the strategy routine of the disk driver and the disk controller itself. An independent management of all these caches and their optimization algorithms may lead to performance degradation, as pointed out by recent research [57, 19].

When importing a remote disk through a CARD driver, the natural question is how to translate the local file system read-ahead policy at the remote disk node, since that node is not aware of the assumptions made by the local system issuing the disk requests. A simple answer is to use a synchronous model employed by local disk drivers (sequentially process the block request queue of the driver) and to rely on the read-ahead facilities of the remote disk controller. But this solution may not work properly for certain classes of applications. For instance, the Web servers serve mostly small files and the read-ahead is really of almost no use in their case. Moreover, the disk fragmentation worsens the performance due to unnecessary disk accesses that pollute the caches and reduce the meaningful disk throughput.

Fortunately, it is fairly easy to translate the read-ahead access pattern to the remote site by sending the disk requests asynchronously. The low-level driver routine concluding the decisions of the strategy procedure removes the currently issued disk request from the driver queue allowing thus the processing of the next requests. The processed requests are gathered in a separate queue that will be walked through when the block replies arrive. Thus, if we consider the low latency of passing a small message over a SAN, we can affirm that the read-ahead requests arrive and are queued at the remote disk node with minimal delay. The experimental section, Section 3.11, quantifies the effects of the asynchronous sending.

Reading ahead raises another question: how much to read ahead? The file system makes its assumptions about the underlying storage system and uses an

upper bound for the read-ahead window. This limit is the maximum read-ahead value of the disk driver (in Linux 2.2, stored in the *max_readahead* array which is indexed by major and minor numbers). If the driver doesn't set this variable, the file system uses a default maximum value. When designing the CARD driver, one has to assess the impact of setting this variable to different values. Intuitively, using a higher read-ahead value would allow for more asynchrony. The intuition is however undermined by our previous claim that the read-ahead block requests arrive at the remote disk node with minimal delay. If so, it means that the CARD driver should use the same maximum read-ahead value as the remote disk driver. Section 3.11.3 will show the sensitivity of our asynchronous mode of sending disk requests to this parameter by acknowledging that the CARD drivers should use the same maximum read-ahead value with the remote disk driver.

3.9 The CARD drivers and the network

The CARD driver design has to cope also with the consequences of using a SAN to move blocks back and forth. The requests arriving at the remote disk site behave like regular messages delivered by the SAN card at interrupt time to the remote host. Handling such messages raises a few issues that we will present in this section. First off, the protocol handler (see Figure 3.3) that processes the incoming block requests may follow either a blocking model (using a kernel thread server) or an interrupt driven one, which serves the blocks using a software interrupt handler run in interruptible context. This handler is executed at the end of the hardware interrupt handler triggered by the SAN card interrupt. Second, messages can get lost. Although SAN interconnects like Myrinet offer high reliability guarantees (erroneously sent packets are seldom, most of the losses are due to insufficiently available buffers on hosts), a virtual disk driver must cope with packet loss events. And third, when the CARD drivers are used together with stand-alone file systems that are unaware of the distributed nature of the underlying storage system, meta-data consistency problems may arise.

3.9.1 Event-driven vs. blocking

The blocking solution uses a kernel thread and has the advantage of establishing a well-defined protection domain which avoids unfairness, as pointed out the by the research experience with network subsystems [11, 12]. It is the thread entity that gets charged for the server computation. The disadvantage is a lower degree of responsiveness. When a request arrives, the network card delivers the interrupt, the server thread is woken up and placed in the run queue. However, making the server thread runnable doesn't ensure immediate response as only the scheduler decides

which thread of control gets the processor next. The scheduling priorities offer a somewhat coarse grain control over when exactly the server thread will run again.

The interrupt driven solution shows better responsiveness. The cached blocks are delivered at interrupt time as fast as possible. If the block is not cached, the software interrupt handler sets up a callback function associated with the buffer. The callback will deliver the block (also in interruptible context) at the time the disk interrupt handler signals that the disk driver finished loading the block. The disadvantage is that the currently running process interrupted either by the SAN card or by the disk interrupt is unfairly charged for unasked for computation.

Our solution is a mixture of the two. The cached blocks are serviced at interrupt time in order to improve responsiveness. For the uncached blocks, the servicing is deferred to a kernel thread because the disk latency is dominant in this case and saving a few microseconds wouldn't help much.

3.9.2 Asynchronous block delivery

The operation of the kernel thread delivering the disk blocks may be synchronous or asynchronous. For the reasons discussed in Section 3.8, a synchronous model is not acceptable. Moreover, a synchronous block delivery mechanism prevents disk strategy routine and disk controller optimizations.

In the asynchronous block delivery mode, the server thread launches a number of disk requests and blocks awaiting for the first request to complete. The benefits arise from amortizing the cost of a context-switch over several disk requests, from the optimizations performed by the strategy routine of the disk driver and from the disk access optimization algorithms implemented by the disk controller. However, there is still one problem left. Once the blocks have been loaded in the memory (either in the local page/buffer cache or, as we presented in Section 3.7, directly in the response socket buffer), they have to be sent back to the requester as well. Sending back the data entails additional processing that cannot be carried out in parallel with the request handling. A possible solution is to use another kernel thread for the block delivery, but then the context-switches and the lack of control on the scheduling decisions increase the servicing overhead even more.

A better solution uses an event-driven model. The disk drivers signal the completion of the block reads by calling a disk software interrupt that removes the requests from the driver queue, marks them free, calls the potential callbacks associated with the buffers heads used for the blocks and launches new disk jobs, if available. Running a callback at interrupt time is the key to better performance, because these callbacks can be used to send the freshly loaded block back to the requester right away. Thus, by simply registering the appropriate callback in the buffer head passed to the strategy routine, one gets the maximum parallelism (actually pseudo-parallelism, as usual on the uniprocessor machines) between the re-

quest processing and the data delivery. The major disadvantage of this solution remains the unfairness, as already pointed out before. The costs of sending the blocks are charged to the currently running process that happened to be interrupted by the disk completion event.

3.9.3 Fault-tolerance

The networks can be unreliable. From a disk driver perspective, a disk request that gets lost is unacceptable. Therefore, although the SANs exhibit usually a high degree of reliability, reliable communication protocols are needed. However, a disk request may be considered lost also when a remote node supposed to serve the request goes down for unknown reasons. A correct (and fault tolerant) solution requires a per-request timer that fires up a handler which flushes out the stale requests, marks the corresponding buffer cache blocks not up-to-date and releases their locks. As presented before, by releasing the locks the application is woken up and the completing system call will report an I/O error.

3.9.4 Meta-data consistency

The CARD drivers are independent of any disk format and may access the physical disk through a raw interface, but using them with a non-distributed file system (Linux *ext2*, for instance) may lead to severe inconsistencies. An example of such problems involves the specialized directory and inode (file meta-data) caches found in any file system. Intended for stand-alone use, such file systems fail to take the appropriate measures for keeping these caches consistent across the cluster when used on top of the CARD drivers. One can enhance the CARD capabilities with file system meta-data consistency mechanisms. Instead, we chose to delegate the job at the file system level for two reasons. First, the CARD drivers are meant to be storage devices independent of the actual data format on the disk. Second, our plans are oriented towards integrating the CARD drivers with Clusterfile [40], a parallel file system for clusters. Clusterfile will take care of the meta-data consistency.

3.10 The CARD operation breakdown

Before we proceed to evaluating the performance of our CARD drivers as a support for sharing data in a cluster-based server, we present in a short experimental preamble a breakdown of the time costs of the CARD remote operations.

3.10.1 Experimental setup and methodology

We ran our experiments on a small Linux cluster consisting of two Intel PCs interconnected through a Myrinet switch and LANai 7 cards. The Myrinet cards have

	CARD	Local Disk
Memory copies	80 μs	-
Service handling	15 μs	-
Network time	205 μs	-
Cached read time	300 μs	30 μs
Disk read time	12300 μs	12000 μs

Table 3.1: CARD vs. Local Disk Comparison (4k block read access times)

a 133 MHz processor on board. They achieve 2 Gb/sec in each direction. The interface to the host is a 64 bit/66 MHz PCI that can sustain a throughput of 500 MB/sec. The Myrinet cards are controlled by the GM 1.4 driver of Myricom [66]. The PCs are 350 MHz Pentium II machines with 256 MB of RAM. All the systems run Linux 2.2.14. The disk used for tests is an IBM DCAS-34330W Fast/Ultra-SE SCSI disk. Only one partition of it, containing approximately 1.7 GB data, was remotely mounted in the experiment that we further describe.

Our experiment attempts to evaluate the performance of a CARD driver acting as a simple remote disk interface. We compare its performance to that of the remote disk it mounts. Essentially, the test consists of running the Unix **find** command to scan a directory for a given string. The directory resides on the remote disk that was mounted by means of a CARD driver. The typical layout of the command was:

```
find <dir> -exec grep <str> {} \;
```

The remote disk is formatted with a native Linux file system format (*ext2*) and because of the potential inconsistencies that we have mentioned in Section 3.9.4, the CARD mount was read-only. The file system is not aged and therefore mostly contiguous. Exclusive caching has not been enforced in this experiment.

3.10.2 CARD vs. Local Disk comparison

The results are presented in Table 3.1. The memory copies represent the sum of the two copying operations taking place at the two nodes involved in the operation in order to transfer the data between the corresponding buffer/page caches and the socket buffers of the two network subsystems. The service handling time refers to the time spent while handling the incoming disk block requests. The network time is computed by subtracting the copying and service times from the total fetch time. Notice that accessing a remotely cached block is significantly faster (40 times, 300 μs vs. 12000 μs) than getting an uncached copy of it from the local disk. Also

interesting is to see that the dominant parts of a remote fetch operation consist of the network time (205 μs) and the copying time (80 μs), respectively. In contrast, the service handling accounts for only 5% of the total access time.

3.11 Performance evaluation

In order to evaluate the performance of our CARD driver as a shared storage for cluster-based Web servers, we used WebStone [61], a well known commercial benchmark for Web servers respecting a Zipf-like [77] document retrieval distribution. The WebStone software has been configured to retrieve static documents only. Therefore, throughout the rest of this section, by WebStone operations we refer to HTTP GET commands. The benchmark used HTTP 1.0 and a file set around 1 GB of data. The server(s) used local disks, NFS (over Ethernet) and Linux *ext2* on top of our CARD driver. The driver used the single copy protocol (that is, it made the disk behave more like a network-attached than a computer-attached storage) and we tested with both of the asynchronous delivery mechanisms: the thread-based one, designated as the *asynchronous* case, and the interrupt-based one, called the *event driven* case.

We drove two kinds of experiments. The first type attempted to assess the performance of a single server using CARD drivers. We evaluated the impact of the load, the disk fragmentation and the read-ahead policy of the file system on the server performance. The results are presented in Subsections 3.11.2 and 3.11.3. The second type of experiments considered the operation of the CARD drivers in distributed servers. The aim was to get an idea about the behavior of our remote disk acting either as a computer-attached or as a network-attached storage. In the first case, the node hosting the physical disk was running a Web server instance as well. For the other case, two server machines mounted locally the remote disk found on a third machine through CARD drivers. Subsection 3.11.4 presents and discusses the corresponding results.

3.11.1 Experimental setup

The cluster setup is that previously presented in Section 3.10.1 with an additional IBM DCAS-34330W Fast/Ultra-SE SCSI disk and another cluster node. Similarly, only disk partitions are remotely mounted for the experiments. Both disks are formatted with a native Linux file system format (*ext2*). One of the disks has an aged file system on it (39.5% non-contiguous, as reported by the *fsck* command), while the other one is newly formatted (0% non-contiguous). The fragmentation factor of the aged disk is not artificially chosen, but has been "naturally" induced by several years of use. The server machines mounted the remote disk partitions read-only (for the reasons explained in Subsection 3.9.4).

As a Web server we used Apache 1.3.20 [5]. A Linux router stays between the client and the server machines. Both the client and the router are Athlon AMD XP 1.5 GHz PCs with 512 MB of RAM and run Linux 2.4.18. The client, the router and the servers are all interconnected through regular 100Mb/s Ethernet.

3.11.2 The impact of the WebStone load on the CARD driver

We varied the load WebStone puts on the server by instructing the benchmark to use a number of simultaneous connections of 150 and 300, respectively. In terms of the requested data, these figures correspond to roughly 600 MB and 1 GB, respectively. The results are presented in Figures 3.6 and 3.7. The average response time refers to the average time taken by an operation (GET command, actually).

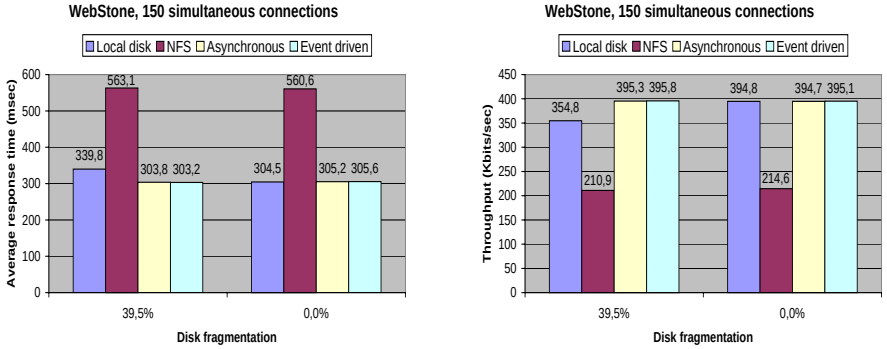


Figure 3.6: Average response time and throughput figures for 150 simultaneous connections

Notice that for the light load (Figure 3.6) there are no significant differences between the two remote disk methods. Naturally, the local disk performance of the aged disk is somewhat worse than that of the non-aged one. Surprisingly, NFS copes better with the disk fragmentation than the local disks. Its insensitivity to the disk fragmentation for this load is similar to that exhibited by the CARD driver cases. Overall, the CARD drivers outperform the local disks and NFS. The difference is unnoticeable for the non-aged disk, but visible for the aged one. This result may be a bit surprising when it comes to the local disks, but it must be recalled that the CARD drivers use a highly asynchronous mode of operation both at the local and at the remote sites. This mode of operation allows some degree of processing overlapping that makes up for the lack of contiguity of the aged disk whose greater mechanical latencies can be thus better hidden.

Under heavy load (Figure 3.7), both of the CARD driver methods yield some sensitivity to the disk fragmentation. Overall, the event driven method clearly outperforms the asynchronous one. The disk fragmentation affects the comparison to

the local disks. For the non-aged disk, the event driven method outperforms the local disks, but the gain is minimal. For the aged disk, as the load increases, the lack of contiguity entails a higher performance degradation for the local disk. Both of the CARD driver methods yield clearly better figures as they manage to hide more of the increased mechanical latencies in the parallel processing of the two nodes. The higher the degree of asynchrony (and therefore the pseudo-parallelism), the better the performance.

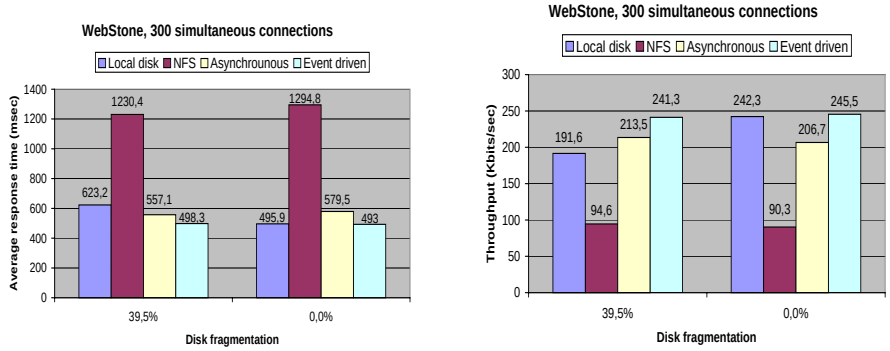


Figure 3.7: Average response time and throughput figures for 300 simultaneous connections

3.11.3 The impact of the read-ahead policy of the file system

As mentioned in Section 3.8, one interesting question is whether the CARD driver exports properly the effects of the read-ahead policy of the local *ext2* file system. To answer the question, we varied the size of the maximum number of the read-ahead pages. We used the event-driven method and a load of 300 simultaneous connections. The results are shown in Figure 3.8. The kernel default value specifies to read ahead at most 31 pages. Notice that for both of the disk types, the CARD driver yields the best performance for the same value. Using smaller or larger values for the maximum read-ahead value of the CARD driver yields a poorer performance than that for the default value. This result shows that the CARD drivers issue read-ahead requests with a minimal delay, since the default value matches perfectly the peak performance of the disk driver at the node hosting the disk. More interesting, for the other read-ahead values, the performance of the aged disk is better than that of the non-aged one. This result points out clearly that exporting a wrong maximum value for the read-ahead window is not only suboptimal, but ceases to serve the purposes of reading ahead (since the contiguous disk performs worse than the non-contiguous one, which is totally counterintuitive). Such decisions can thus affect the performance of the remote system and therefore undermine our design

goal to minimize the remote impact of the policies that are run locally.

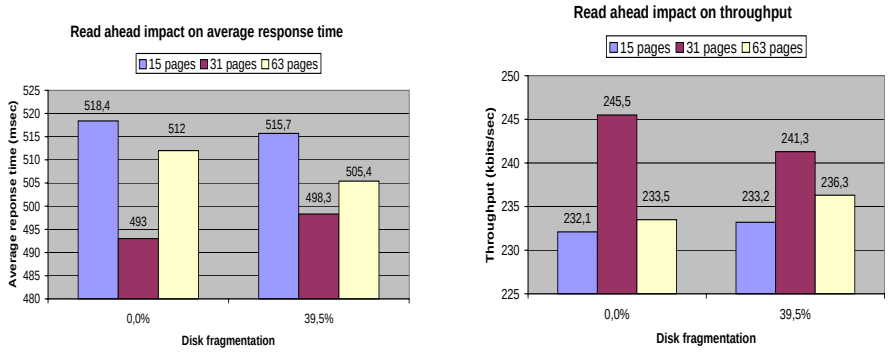


Figure 3.8: The impact of the file system read-ahead policy

3.11.4 Distributed server performance

The distributed server performance evaluation uses two server machines in three setups. The router acts as a front-end dispatcher using a Round Robin request routing policy. In the first setup, the two machines serve the requests from the local disks (the same file set, replicated on both disks). The disks are mostly contiguous (0% and 1.7% non-contiguity, respectively). In a second scenario, one of the two servers uses a virtual disk driver to mount the 0% non-contiguous disk locally using the event driven method. This case corresponds to the computer-attached operation of the disk. In the third setup, two machines mount locally the remote disk as a network-attached disk using the event driven method. The WebStone load was 300 simultaneous connections. The results are reported in Figure 3.9.

The two servers equipped with local disks perform best. The load is almost equally split between the two disk drives and that maximizes the disk parallelism. The other two cases show that the task is disk and not computational dominated. Indeed, notice that their performance doesn't differ much from that of the corresponding single server in Figure 3.7 (slightly worse for case two and even better for case three). For the second scenario, we assume that the additional load placed by the server software on the node hosting the disk is responsible for the light performance degradation. To conclude, the disk utilization is affected by the additional remote operation.

Another important observation is that the performance of the virtual disks doesn't scale well when compared to that of a simple technique like replication. A two-node distributed server using replication performs significantly better than one using our CARD drivers. This observation leads to the conclusion that one needs improved caching techniques to bridge the performance gap between the

replication- and the virtual disk-based solutions.

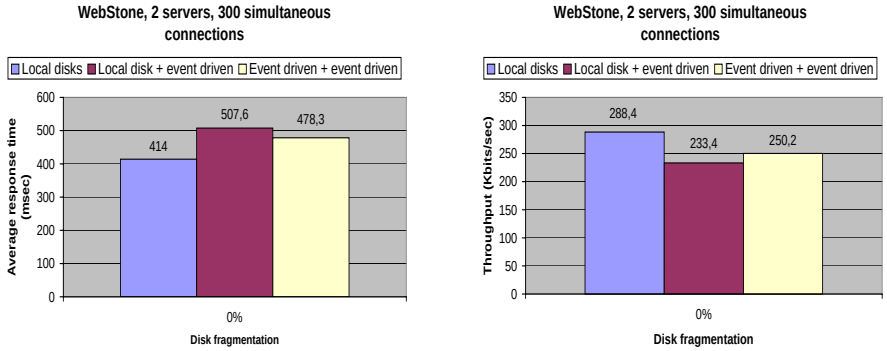


Figure 3.9: Distributed server performance, 300 simultaneous connections

3.12 Summary

In this chapter we presented the design of our CARD driver, a block device driver in the kernel virtualizing the accesses to the remote disk data over a SAN in COTS clusters. The main contributions of this construct come from its design that attempts to translate the SAN bandwidth and latency figures into maximal block retrieval performance figures. For instance, accessing a remotely cached block through the CARD drivers is roughly 40 times faster than accessing the local disk (when the block is uncached). A dual, both page- and block-oriented operation of the driver as well as a single copy protocol that implements exclusive caching enable a high performance control over the memory usage. The single copy protocol is possible as we developed mechanisms to perform DMA from the disk into the socket buffer (that is, by bypassing the page/buffer cache). To the best of our knowledge, this mechanism is a feature unique to the CARD drivers as no other virtual disk drivers offer similar capabilities. When used to implement exclusive caching, the single copy protocol makes the remote storage look like either computer- or network-attached storage. The CARD driver exports properly the local file system read-ahead decisions to the remote disk system through a highly asynchronous mode of operation that uses the SAN optimally. The optimality of the SAN usage is proven by the fact that the CARD maximum read-ahead value coincides with that of the remote disk driver. In fact, the CARD-issued disk requests arrive at the remote disk node with a minimal delay. Also, by overlapping the communication with the request processing, the highly asynchronous operation mode yields a performance comparable to that of the local disks.

Chapter 4

CARDs and cooperative caching

In the previous chapter we presented the software architecture of the CARD drivers and we evaluated their performance as a shared storage for cluster-based servers. However, the evaluation results presented in Figure 3.9 show that our virtual disks are outperformed by a simple replication-based technique and that points to the need to improve the performance of the CARDs through extensive caching. We chose to rely on cooperative caching [23] in order to achieve this goal and this chapter is devoted to the cooperative cache-enabled operation of the CARD drivers.

As seen in the previous chapter, a node mounting locally a remote disk by means of a CARD driver sends block requests over the SAN. Once arrived at the disk node, the requests are checked against the local page/buffer cache. Many CARDs mounting the same remote disk develop a star-shaped cache topology centered at the disk node. This topology allows a limited form of cooperative caching as consecutive requests to the same disk block benefit of the cached copies of the block stored in the page/buffer cache of the disk node, provided that such copies haven't been evicted meanwhile. The latter assumption is the starting point for a more complex solution involving the caches of all (or just some of) the nodes in the cluster. The workloads exceeding the local node memory would definitely not take advantage of the caching capacity of the disk node. A possible solution would be to save the evicted blocks in remote memories. Extending the caching capacity beyond the local memory by implementing a joint management of the cluster caches represents the topic of cooperative caching and this chapter. The basic premise behind the idea is that getting cached copies of a block from remote memories is faster than accessing the disk (even if the disk happens to be local). Our CARDs fulfill the prerequisite as it can be easily deduced from Table 3.1.

As mentioned in Section 3.1, cooperative caching was introduced by the serverless file systems [23, 4] which extended the traditional distributed file system mem-

ory hierarchy (client cache, server cache, server disk) by adding the extra level of the remote client memories. Thus, misses in the client or the server cache can be checked against the remote client caches as well before going to the server disk.

Our approach tackles cooperative caching at the storage system level. Instead of reasoning about a file system hierarchy, we prefer to view cooperative caching as a way to enlarge the disk cache by using cooperative remote caches. Thus, the upper layers are offered the view of a unified buffer/page cache across the cluster. This approach appears to us more general, as one can build on top of our storage system not only file systems but also databases or Web storage systems. The CARDs are storage-level kernel constructs that extend the local disk caching capacity beyond the locally available memory by relying on cooperative caching policies. These can be downloaded in the driver at will by the user applications, akin to grafting/extensible kernels [28, 14]. Such a policy is supposed to define a distributed algorithm for the management of the individual cluster caches and, by doing so, it builds a global cooperative cache. The capacity of this cache can grow up to the size of the global cluster memory provided that the appropriate cluster-wide block lookup and retrieval procedures are available. A cooperative caching policy defines also a global replacement algorithm for the locally evicted blocks.

The cooperative caching policies are independent of the CARD driver they use. A collection of CARD drivers using a common policy on several cluster nodes results in a unified page/buffer cache (the aforementioned “cooperative cache”) across the participating nodes. A CARD driver making no use of cooperative caching behaves in fact like a remote disk interface (as described in the previous chapter) and is restricted to a peer-to-peer relationship to its corresponding disk.

The CARD drivers offer a flexible and easy to use scheme of building the cooperative cache. Just by mounting CARDs for a given disk on a set of cluster nodes and by downloading the appropriate cooperative caching policy in each of these CARD drivers, one gets a share of each page/buffer cache in the set of the cluster nodes for cooperative use. The span of the cooperative cache is thus determined by the number of the participant nodes and their corresponding buffer cache shares.

Managing the cooperative cache is a task orthogonal to that of physically moving blocks to and fro and is expressed by means of a cooperative caching policy. Such a policy defines how to find a certain block in the cooperative cache, what should be done with the evicted blocks, whether multiple copies of the block may exist in the cooperative cache and, if so, how consistency is handled. The blocks not found in the local caches can now be searched for in the cooperative cache. Only when there is no available copy in the entire cooperative cache, the block request is serviced from the disk.

This chapter presents the cooperative caching implementation within the CARD driver as well as two cooperative caching algorithms, one proposed by Dahlin et

al. [23] and another one developed by us. Our algorithm is called Home-based Server-less Cooperative Caching (HSCC) and aims to provide for a decentralized, cluster-wide cooperative cache. We evaluate the two algorithms in the context of a distributed environment by using a micro-benchmark devised by us. An evaluation of HSCC and the CARDs in the context of the cluster-based Web servers follows in Chapter 5. For a discussion on the use of the cooperative caching to improve the performance of the collective I/O operations in the Clusterfile parallel file system, we refer the reader to Isaila et al. [39].

An abbreviated version of the material presented in this chapter was previously published in [48].

4.1 Related work

The xFS project [23, 4] introduced the notion of cooperative caching. HSCC resembles *Hash Distributed Caching* [23] as both are server-less and use a hash function on the block number. Also, HSCC shares with the *N-Chance Forwarding* [23] eviction algorithm the notion of *recirculation count*, called *depth* in HSCC, although used in different context. The *recirculation count* applies to single cached copies of a block (singlets) while *depth* is used for the most recent block copy. *N-Chance Forwarding* and HSCC handle eviction differently. *N-Chance Forwarding* chooses randomly a node to forward an evicted singlet. Instead, HSCC takes an informed decision by sending the block to the node which appears to be the least loaded one from the local perspective. Since HSCC's knowledge about the block copies is distributed and not centralized as in *N-Chance Forwarding*, sometimes there might be some unnecessary block forwarding.

PACA [21] is another cooperative file system cache that uses a decentralized algorithm. It attempts to avoid replication and the associated consistency mechanisms by allowing only one copy of the block in the entire cluster-wide cache. Keeping a single cluster-wide copy is possible as PACA uses a *memory_copy* mechanism (a sort of Remote DMA) to send the data from the global cache to the local user memory. However, every data access has to go through this *memory_copy* mechanism which is clearly much slower than accessing a local block copy.

Sarkar et al. [55] describe another cooperative caching algorithm using hints. Like HSCC, it tries to avoid centralized control. Reasonably accurate hints are used to locate a block in the cache without the involvement of the server. The algorithm defines a master copy of each block and the clients exchange hints about the possible location of this copy. If these hints are not accurate enough, there is a fall-back mechanism that gets a copy of the block. The master copy simplifies the eviction: only the master copies are subject to a saving attempt. The algorithm uses a "best-guess" replacement strategy when storing locally the remotely evicted

blocks. On the contrary, HSCC relies on the local kernel policy to do that. However, the "best-guess" replacement may introduce erroneous decisions and these are offset by adding an extra cache, the *discard cache*, at the server node. The cache consistency is file-based and not block-oriented as in HSCC.

The Global Memory System (GMS) [29] is a distributed shared-memory system that can implement cooperative caching. GMS uses managers to locate the blocks in the client caches and doesn't provide a consistency mechanism. Only single cached copies of a block are saved in the cooperative cache. A centralized algorithm chooses the client target for the evicted block by periodically collecting age information of the blocks in the client caches. The location of the oldest block is then communicated to all the nodes.

We mentioned in Section 3.1 that the SIOS [36] system uses Virtual Device Drivers (VDD) to amass the entire disk capacity of a node (both local and virtual disks) similar to a RAID system (actually implemented as the default technology). The SIOS system implements also cooperative caching, but neither a description of it nor its evaluation is provided. Our design was oriented towards individual disks that use a share of the cooperative cache according to their own policies. When compared to the CARDs, the VDDs offer a much less flexible way to partition the cluster caches and lack a mechanism for changing cooperative caching policies.

4.2 Bringing the CARDs and cooperative caching together

Every miss in the local buffer cache is checked by the CARD driver also against the remote page/buffer cache at the disk node. As several nodes mount locally a remote disk through a CARD driver, the global caching system develops to a star topology, centered at the buffer cache of the disk node. This central buffer cache plays the role of a server cache queried by the other caches through their corresponding CARDs. However, this topology cannot serve the purposes of cooperative caching as defined by Dahlin et al. [23]. In order to take advantage of the whole cluster memory, one needs to implement a distributed management of the individual caches across the cluster.

The CARD drivers support cooperative caching by enabling the applications to download into the driver their own joint management algorithm of the cluster caches (naturally, it is assumed that all the nodes mounting a certain remote disk will use the same algorithm). Downloading policies into the driver is accomplished by providing hooks in the CARD driver code. The policy is expressed as a kernel module implementing the appropriate hooks.

One can see in Figure 4.1 how cooperative caching changes the behavior of a CARD driver acting as a simple remote disk interface. With cooperative caching,

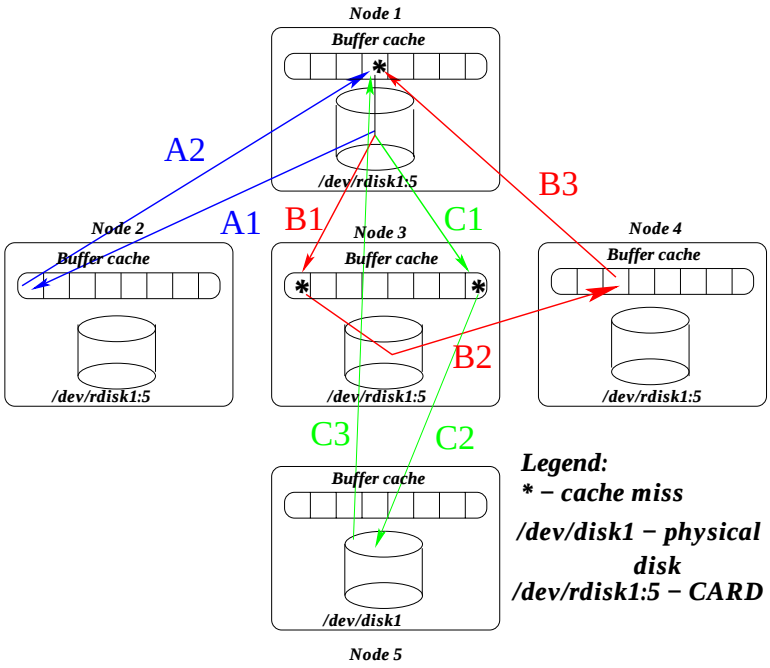


Figure 4.1: Cooperative caching with CARDs. **Case A**: client-to-client cooperation; **Case B**: three-client cooperation; **Case C**: client-to-client cooperation fails. The block must be retrieved from the disk

when a block request misses in the local cache, the appropriate hook in the CARD driver redirects the request according to the local knowledge. A first possible scenario (Case A in Figure 4.1) assumes that the requesting node (Node 1) has local knowledge about where the requested block might be. As a result, it will direct the request to the node it suspects to have a block copy (Node 2). Upon receiving the request, Node 2 replies by sending back a copy of the block. This is the ideal case, when fetching a copy of the block entails only two messages and two copy operations (between the buffer cache and the network buffers at the two nodes).

In a second scenario (Case B), Node 1 uses a local hint to route the block request to Node 3 but the hint proves wrong. Node 3 doesn't have anymore a cached copy of the block but knows that Node 4 must have one and thus forwards the request to that node. Node 4 replies directly to Node 1 with the block copy. The direct response saves an extra network hop that otherwise might prove costly, both in terms of interrupt and network latency (a typical disk block size is 4 KB).

The third case (Case C) depicts the fall-back mechanism that ensures the eventual block delivery even when there is no cached copy available. Node 1 sends

the request to Node 3 but this one neither caches the block nor does have any hint about where the block might be. Therefore, Node 3 decides to forward the block request to the disk node (Node 5). Node 5 accesses the disk and delivers a copy of the block to the requester (Node 1).

The advantages of the cooperative caching become now clearer. By extending the caching capacity potentially to the size of the global memory of the cluster, one can use working sets that scale beyond the local memories. Moreover, the disk node performing the uncached disk I/O gets offloaded. Also, as already mentioned, accessing remote memories over the SAN improves the read latency.

In order to be able to yield such a cooperation, the CARDs must offer a clear interface to the applications. Designing such an interface needs identifying the main operations in the the above scenarios. This is the subject of the next section.

4.3 Cooperative caching policies

A cooperative caching policy is a set of four operations executed by the CARD drivers in response to various events during the lifetime of a disk block in the cooperative cache. They form a distributed management algorithm for the cooperative cache. Writing a cooperative caching policy means to provide code for these operations in a kernel module and to hook them up with a CARD driver. By default, their body is void and then the CARDs behave like remote disk interfaces. The C definition of a cooperative caching policy looks like this:

```
struct coop_caching_ops {
    int    (*lookup)(struct request*);
    int    (*handle_eviction)(struct
                      buffer_head*);
    int    (*keep_consistency)(kdev_t,
                     struct sk_buff*);
    kdev_t (*handle_request)(kdev_t,
                     struct sk_buff*);
};
```

4.3.1 Block lookup

Lookup is a typical client-side operation. The clients call *lookup* implicitly on a cache miss as part of the low-level strategy routine of the CARD driver, right before sending the block request over the SAN. Every time that a block request misses in the local cache, the CARD driver needs to find out where to forward the request. Searching the cooperative cache for a block copy is the job of *lookup*. It returns either the ID of a node caching the block or the ID of the disk node.

However, depending on the policy designer, *lookup* may very well be called also by the physical disk driver when the block requests miss in the buffer cache (assuming that there are cached copies of those blocks somewhere else). Such cases correspond to serving the server cache misses from the remote client memories before going to the disk. In this case, *lookup* is called by the low-level disk driver strategy routine.

4.3.2 Handling the eviction of cached blocks

Cooperative caching tries to reduce the I/O activity by increasing the cache hit ratio. This goal can be supported not only through cluster-wide block lookups but also by using idle remote caches to store the locally evicted blocks. Especially the single cached copies of a block (*singlets*) should face a special treatment. Discarding such blocks from a buffer/page cache means that the next request for that block anywhere in the cluster will have to go to the disk.

handle_eviction looks for remote caches that can host locally evicted blocks. However, it does not interfere with the local eviction algorithm. The CARDs are considered ordinary block devices and they obey the kernel native eviction policy as any other local disk would do. *handle_eviction* extends this algorithm by using the cooperative cache and implements a distributed eviction algorithm (and indirectly, a global, cluster-wide cache replacement policy). If *handle_eviction* succeeds, it returns the node ID of the new block host. If it fails, the block is dropped.

Technically, the block eviction works as a hook in the clock algorithm of the kernel memory management subsystem. This is the only kernel code change needed for our CARD-based system. In Linux 2.2, this change affects the code of the *shrink_mmap* routine that implements the clock algorithm. Our hook is called exactly before releasing the memory page storing the disk block. The code of *handle_eviction* chooses a target for the block copy, copies the page into a socket buffer and sends the buffer content over the SAN to the designated host.

4.3.3 Consistency issues

Cooperative caching is a technique that may use replication to improve the read latency. In such cases, the policy should deal with the inconsistencies introduced in the cooperative caching by the writes. We use the term consistency and not coherence as we refer to writes to different memory locations and the order in which they are noticed by the other nodes in the cluster [65]. Each write operation of a cooperative caching policy using replication triggers the execution of the corresponding *keep_consistency* routine. Essentially, as soon as the low-level routine of the CARD driver at a client node decides where to send the written block, *keep_consistency* fires up and triggers the execution of the consistency protocol.

Typical consistency algorithms include *invalidation-* or *update-based* [65] protocols. Invalidation protocols send invalidation messages to all the copies of the written block. When the holder of an invalidated copy needs to access it again, it will have to get the new version from someone else (presumably the holder of the written copy or the disk). Since the invalidation messages are small, the additional network traffic is reasonably low but the future accesses to the updated data need to go over the network. Update protocols propagate the update to all the copies of the written block. The update propagation speeds up the future accesses to the updated copies but may generate a lot of network traffic.

4.3.4 Handling block requests

When receiving a block request from another node, a node has to know how to handle it. When the CARDs act as simple remote disk interfaces, this job is that of the protocol handler in Figure 3.3. Its operation has been described in Chapter 3. However, when the cooperative caching is in use, a block request may hit another node than the disk node, as seen in Figure 4.1. Therefore, the basic operation of the protocol handler has to be enhanced with cooperative caching aware operations.

handle_request is such a cooperative caching aware operation. It decides whether the block requests can be satisfied from the local cache or not. If a local copy exists, the method returns the corresponding local CARD ID (or the disk ID, on the disk node) and the block is serviced. If not, the requests are forwarded to other hosts. Choosing the forwarding target is based on locally or globally available information about the block copies across the cluster. If no such information is available, a conservative approach is to forward the request to the disk node. An error code should be returned to the requester when the forwarding is not possible.

handle_request defines also how to deal with the requests to save remotely evicted blocks. The idea is to make room locally to store the incoming block if enough memory is available. Otherwise, the routine may decide to send the block further to some other node potentially willing to store it.

Special care must be taken when writing *handle_request* because it is being called by the protocol handler of the CARD driver (see Figure 3.3). Since this one is executed in interrupt context, all its code is prone to race conditions with the main kernel thread.

4.4 HSCC: Home-based Server-less Cooperative Caching

According to the above guidelines, we designed Home-based Server-less Cooperative Caching (HSCC), a decentralized, globally coordinated cooperative caching

policy which attempts to offload the disk node by distributing the block servicing task. HSCC is *home-based* because each block gets a *home* node that services requests for that block. The homes are assigned to the blocks by using a simple hashing *modulo n* scheme. Since usually the clusters don't change dynamically in size, the choice of a fixed *n* is reasonable enough. The home may cache the block in its own buffer cache or may simply keep a hint telling which is the node caching the most recent block copy. For the latter case, the home forwards the block request to the node specified by the hint. The blocks get loaded at their homes lazily (on-demand) and only if there is enough room. Thus, HSCC partitions statically and evenly only the meta-data, not the actual data. HSCC is not fault-tolerant, the failure of a home node breaks the operation of the algorithm. The policy is called *server-less* because the disk blocks are not serviced by a centralized server.

HSCC employs a per node cache index to keep track of the nodes caching particular blocks of the disk. The node uses this index to forward the block requests that miss in the local cache. In general, keeping indices for cached blocks may be too space-expensive and may endanger the scalability. For instance, if the index entry uses a bitmap to mark the nodes caching the block, keeping track of 128 nodes would require 16 bytes per entry just for that. 1024 nodes would require 128 bytes per entry. That means that the local extra memory consumption grows steadily with the size of the cluster and this is a serious scalability problem. For example, 1024 nodes equipped with 256 MB each build a 256 GB cooperative cache. 128 bytes for every 4KB block means 3.1% of the indexed memory. But 3.1% out of 256 GB is already too much for a single node. And the things don't improve for a linked list implementation.

To avoid this inconvenience, we choose to keep the index entry size constant, regardless of the cluster size. We store only two node IDs per index entry. When storing a block, a node records in the corresponding index entry the ID of the node delivering the copy. We call this ID the *previous* ID. The second ID is that of the last node that requested the block from the local node. We call it the *next* ID. Now all the block copies are chained in a double-linked list in which *next* points to the more recent block copies while *previous* refers to the older copies. This solution doesn't reduce the overall size of the index, but distributes its information and thus lowers the local memory usage.

The overall index size can be reduced significantly if the block size (or, equivalently, the cooperative caching unit) is increased to values larger than 4KB (a typical value for SCSI disks, for instance). But larger caching units aggravate the *false sharing* problem pointed out by the distributed shared memory research experience. False sharing designates non-overlapping concurrent accesses to a shared page of memory. Therefore, although the nodes accessing the page do not refer the same part of the memory page, they still have to obey consistency protocols, for

instance, and, by doing so, the system can experience severe page thrashing. The larger the page, the higher the probability of false sharing.

4.4.1 HSCC_lookup

When a client node misses in the local cache, the CARD driver must find a node to which to send the block request. The low-level part of the strategy routine of the driver looks up the local cache index for a valid *next* ID by means of *HSCC_lookup*. If found, the routine returns this ID and the CARD driver directs the request to the node identified by it. The rationale behind this procedure is that the *next* node is the last one to have asked the local node for the block and there might be a chance to find there a copy of it. If no such hint exists, the request is sent to the home which is supposed to have accurate information about the last accesses to the block across the cluster.

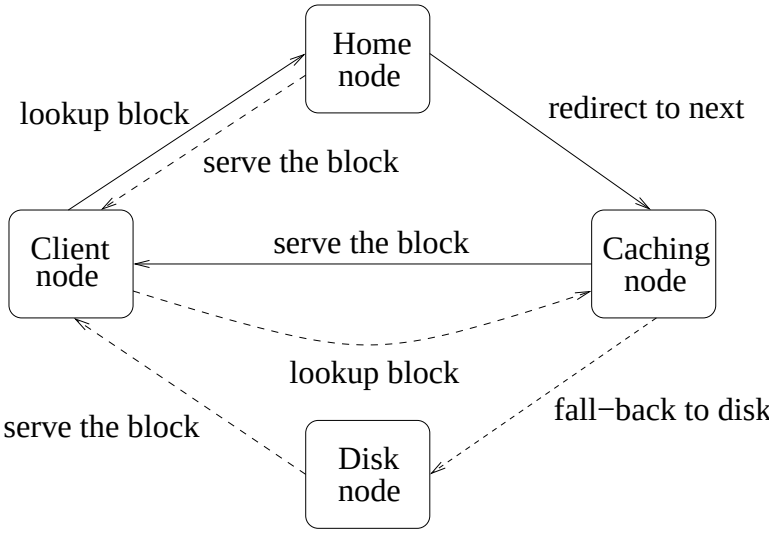


Figure 4.2: Block retrieval in HSCC

4.4.2 HSCC_handle_request

If a block request arrives at its home, the protocol handler (see Figure 3.3) calls *HSCC_handle_request* which looks up a block copy in the local buffer/page cache. If found, the block is returned to the requester. Otherwise, the request is forwarded to another node that can satisfy it. This node is either that identified by the *next*

ID from the corresponding index entry of the home node, if any, or the disk node, otherwise. Either way, the requester is registered in the home index as the *next* ID.

If the block request arrives at a node that is not the home of the requested block, it means it is a forwarded block request. Such requests are handed over to *HSCC_handle_request* which checks if the node holds a block copy. If so, the copy is delivered directly to the original requester of the block in order to save network bandwidth and to avoid an extra and unnecessary network hop. Otherwise, if this node is not the disk node, the request is forwarded directly to the disk node. In turn, the disk node will deliver a copy of the block.

Figure 4.2 visually summarizes the combined operation of *HSCC_lookup* and *HSCC_handle_request*.

4.4.3 HSCC_handle_eviction

An evicted block is considered a singlet if the *next* ID is void and the *previous* ID is the disk node ID. Otherwise said, if this copy of the block was taken from the disk (here works also the assumption that the exclusive caching is in place by default) and not given to anyone else so far. For a void *next* ID and a *previous* ID different than the disk node ID, the block is considered the most recent copy. That is, this block copy has been taken from some other client cache but not yet given to any other requester. It may actually be a singlet if all the other copies have been evicted meanwhile. Only singlets and most recent copies are considered for saving in remote memories. All the other locally evicted copies are simply discarded.

Singlet eviction

HSCC_handle_eviction sends a singlet to the least loaded node in the cluster, as perceived from the local perspective. Choosing such a node is based on a priority queue storing the numbers of the blocks cached on behalf of each home participating in the cooperative cache. The home on behalf of which the local node caches the least number of blocks is considered the least loaded node. If the chosen node can host the block, it will send an index update with the new *next* ID to the home of the block. If not, the target node forwards the block further to its home. If there is no room at the home either, the block is discarded.

Most recent copy eviction

Most recent copies are sent to their *previous* node. The rationale behind this decision is that the *previous* node might still have a copy of it and thus no saving operation is needed. If the *previous* node has a copy, the block is discarded. Else,

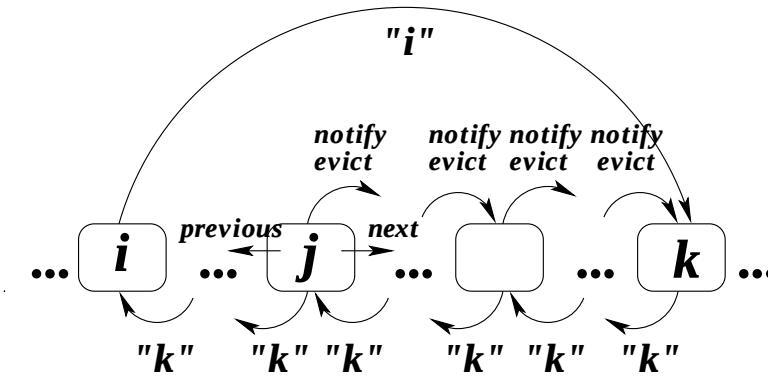


Figure 4.3: Eager cache index entry elimination algorithm. Node j evicts a block and triggers the algorithm that will flush the corresponding cache index entry

if there is enough space, the block is stored and an update message is sent to the home with the node ID as the new *next* ID of the home of that block. If there is not enough room, the *previous* node forwards the block further to its own *previous* node.

In practice, this process can be quite long, so a *depth* count set in the evicted block message helps restrict the forwarding to *depth* stages. If no copy is found in *depth* steps, the block is stored locally instead of a "non-CARD" block in order to avoid an eviction ripple effect that would trigger another *HSCC_handle_eviction* operation, this time for the newly evicted block. By default, *depth* is 2.

When the index update message reaches its home, if there are no newer block requests, an invalidate index message is sent to the evicting node. This node discards its cache index entry and forwards the invalidation to its own *previous* node. Each *previous* node does the same until the invalidation reaches the node caching the new most recent copy.

Discussion

In the most recent copy eviction algorithm, a cache index entry is kept around (even if its corresponding block has been evicted) until the node receives an index invalidation message from its own *next* ID node. This solution favors a local and simple decision over a global and more complex one that would attempt to eagerly update the *next* and *previous* ID nodes at the block eviction time.

The eager solution simplifies a lot the most recent copy eviction because it uses a consistent mapping between the index entry and the block it indexes. Since the list is up to date, the most recent copy is discarded. The home is updated with the

previous node as the new most recent copy holder (i.e., the new *next* pointer at the home).

This solution is sketched using the annotations from Figure 4.3. When a node *j* evicts a block, it informs its *next* ID node. If *next* evicts the block too, it informs its own *next* ID node. As soon as this info reaches a node, say *k*, that doesn't want to evict the block, *k* sends back to its own *previous* ID node a message stating that he is the new *next* ID node. *k* marks its own *previous* ID node as invalid and waits for an update. The message issued by *k* travels back until it finds the first node, say *i*, that doesn't evict a block copy. At each hop on his way to *i*, the message sent by *k* determines the node to discard the corresponding cache index entry. When *i* receives the message, it modifies its own *next* ID node and sends an update directly to *k* with the new *previous* ID node. In turn, *k* will update its own *previous* ID node.

4.4.4 HSCC_keep_consistency

Since HSCC uses block replication, the consistency becomes an issue. The consistency algorithm is a flavor of write-through with invalidation. A written block is sent to the disk node and both the home and the *previous* ID node copies are invalidated. The home forwards the invalidation to the node caching the most recent copy. Each node receiving an invalidation message invalidates its own copy. If the *previous* ID node is not the disk node, the invalidation is forwarded to it. Otherwise, the message is dropped, since the disk node already got the written block so the invalidation would be wrong. As noticed, there are no guarantees for concurrent writes. Concurrency control would need a locking scheme in the upper kernel layers (at the file system level).

4.5 On the scalability of HSCC

One of the main advantages of the COTS clusters is their inherent scalability as new PCs can be steadily added to the cluster to cope with increasing computational demands. Whether the software running on clusters is scalable is another matter. We have not had the chance to run HSCC on large scale clusters to prove experimentally its scalability. Therefore, this subsection attempts to shed some light on the theoretical aspects of the algorithm that argue in favor of its scalability and on those that raise questions.

As previously seen, a block request message can travel over at most two hops in the worst case. Indeed, if the requester doesn't have a hint of where a copy of the block might be, it will ask the home. If the home has a record pointing to a node, the message gets forwarded to that node. However, if the designated node doesn't

hold a copy, it uses the fall-back mechanism and sends the block request to the disk node. Altogether, this procedure sums up to a constant number of messages, namely four, and thus renders the block retrieval independent of the cluster size.

Moreover, the caching and the meta-data management in HSCC can only benefit from larger clusters. It is trivial that a larger cooperative cache will accommodate better the caching needs of a given disk remotely imported through CARDs. Beyond that, the meta-data management of HSCC will benefit too because it distributes the responsibility of handling the block requests in equal shares to the homes. Since a disk has only a given number of blocks, the larger the size of the cluster (i.e., the number of homes), the smaller the number of the blocks that a given home has to take the responsibility for.

The eviction handling is also taken care of so that it doesn't hurt the scalability. As previously mentioned, there is a constant depth (a constant number of hosts tried) that an evicted block has to go through. This measure was first thought against an increased eviction handling overhead, but it perfectly matches the scalability requirement as the number of saving attempts concerning an evicted block is independent of the cluster size.

All this sounds like good news, but a thorough analysis needs to look at the home operation as well. When no local hints are available, HSCC looks up blocks based on a hash function on the block number that yields the home ID. It is not unconceivable that, for a certain workload, all the nodes in the cluster (in the worst case) ask the same home for the same block (or for different blocks with the same home). While each of the requests will incur a constant number of message exchanges as seen before, it is clear however that the operation of that given home is overwhelmed by the number of messages coming from the cluster. And this number varies now with the cluster size. The situation is serious as each of the incoming messages triggers an interrupt and thus inflicts increased interrupt latency, a known cause for poor system performance. As of now, we do not have a precise evaluation of this kind of behavior, but we believe it to be an important issue.

4.6 HDC: Hash Distributed Caching

We implemented also one of the algorithms presented in [23] as *Hash-Distributed Caching*, from here on designated as HDC. We chose HDC over *N-Chance Forwarding* and *Globally Coordinated Caching* [23] because it is fully distributed, whereas both the aforementioned algorithms redirect the client requests through the server (or specially designated managers). Since the clusters are more likely to be used as parallel machines (as opposed to the distributed environment in which cooperative caching has been used first), we believe this feature of HDC to be an

important aspect as routing all the block requests through the server may become a bottleneck of the system. HDC has been reported to perform similarly to *Globally Coordinated Caching* and close to *N-Chance Forwarding* [23].

HDC employs also the notion of a home. The home of a block is that node in the cluster whose ID matches the result of applying a modulo hash function on the block number. The HDC home, unlike the HSCC homes, is supposed to hold a block copy. If this is not the case, the home redirects the block requests to the disk. Here is HDC's description in terms of our cooperative caching policy definition.

- *HDC_lookup* - This operation simply redirects all the local cache misses to the home of the block. It doesn't make use of any hints (like HSCC) that might help find a block copy somewhere else in the global cooperative cache.
- *HDC_handle_request* - Handling requests is equally easy. If the block is cached, it is simply delivered. Otherwise, the block request is re-routed to the disk node.
- *HDC_handle_eviction* - HDC handles evictions only at the disk node. The other nodes participating in the globally coordinated cache do not implement any *handle_eviction* operation. At the node hosting the disk, the eviction handler simply sends the evicted block to its home in the cooperative cache.
- *HDC_keep_consistency* - The written blocks are sent to the disk node which, in turn, invalidates all the other copies.

4.7 Performance evaluation

We evaluated the performance of our CARD prototype using HSCC and HDC. A disk formatted with a native Linux file system format (*ext2*) was remotely mounted by means of CARD drivers. Because of the potential inconsistencies that we have mentioned in Section 3.9.4, the mount was read-only. The file system was not aged and therefore mostly contiguous. All the tests consist of running the Unix *find* command on a CARD driver to scan a directory for a given string. The typical layout of the command was:

```
find <dir> -exec grep <str> {} \;
```

4.7.1 Experimental setup

We ran our experiments on a 3-node Linux cluster using the machines and the network infrastructure described in Subsection 3.10.1. The experiments use only a 1.7 GB partition of the IBM DCAS-34330W Fast/Ultra-SE SCSI disk that was remotely mounted by means of CARD drivers.

We approximated the extent to which the buffer cache of Linux can grow by scanning a directory whose size was larger than the local memory. On the disk node, the buffer cache grew up to 240 MB. On a CARD node, the buffer cache grew up to 225 MB. So we can consider a value of roughly 690 MB of RAM for our cooperative cache. However, this figure is just an upper bound as the Linux memory management algorithm trades off dynamically application memory for kernel memory. This behavior makes it hard to determine precisely the buffer cache size, which may vary significantly depending on the machine load.

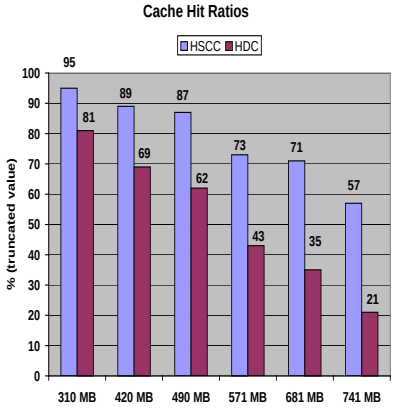


Figure 4.4: Cache hit ratio comparison

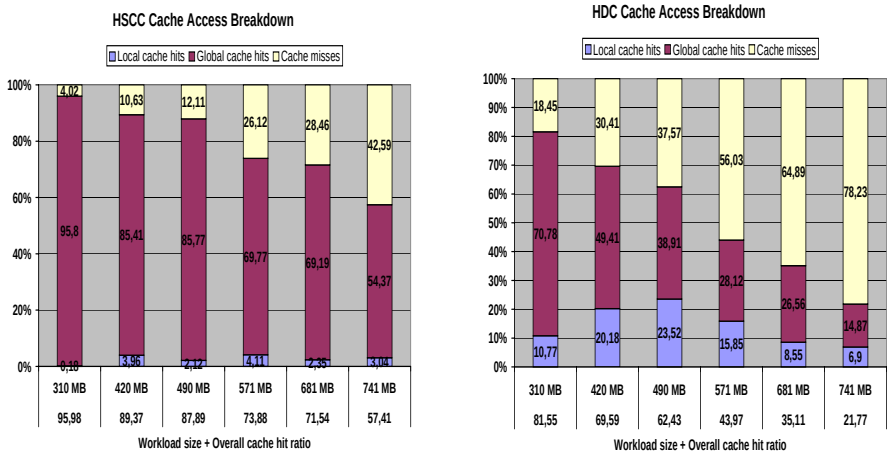


Figure 4.5: Cache Access Breakdowns. Local cache hits, global cache hits and cache misses for HSCC and HDC

4.7.2 CARD operation analysis

The cache-cooperative operation of the CARD driver was evaluated using six workloads whose sizes were roughly 310 MB, 420 MB, 490 MB, 571 MB, 681 MB

and 741 MB, respectively. All the workloads consisted of scanning combinations of subdirectories of a typical */usr* Unix directory, namely */usr/bin*, */usr/lib/*, */usr/share*, */usr/src*. The first four size choices aim at evaluating the behavior of the cooperative cache for workloads bigger than any cluster node memory (240 MB) but smaller than the size of the global cache (690 MB). The last two size choices intend to show the system performance at the limit of the global cache and beyond.

We warmed up the cooperative cache by running the *find* command at a CARD node and then we took measurements by running it at another CARD node. We compared the cache hit ratios of the two policies and evaluated the general benefits of saving the evicted blocks.

4.7.3 Cache hit ratios comparison

A comparison between HSCC and HDC in terms of cache hit ratios is presented in Figure 4.4. Notice that HSCC performs better overall, while HDC's performance degrades faster with the increasing size of the workloads than that of HSCC. Indeed, for the first workload (310 MB), the cache hit ratio of HDC is roughly 85% of the HSCC figure, while close to the global cache limit, that is, for the 681 MB workload, HSCC achieves at least twice as many cache hits as HDC. HDC's degradation becomes even more severe beyond the global cache limit, as for the last workload HSCC's ratio is roughly 2.7 times that of HDC.

Figure 4.5 offers more insight on the CARD operation by showing the cache access breakdowns. Notice that HDC has better local hit figures while HSCC yields better global hit ratios. As soon as the workload approaches the limit of the global cooperative cache (690 MB), the local hits become less important than the global ones and this fact explains the difference in performance that we saw in terms of overall hit ratios (see Figure 4.4). Moreover, the difference in the global cache hit ratios between the two algorithms shows poor HDC eviction handling. The next subsection further clarifies this point.

4.7.4 Eviction statistics

The number of evicted blocks stored by a CARD node running on the warm cooperative cache is reported in Figure 4.6. From this figure and from Figure 4.5, one can infer that handling too many evicted blocks is a waste. The reason lies in the way the two policies handle the cooperative cache. HDC evicts blocks only at the server cache (disk node) and does it irrespective of the cluster load by sending the evicted blocks to their homes. As Figure 4.5 shows, this feature of HDC improves the local cache hits. Overall, HDC has better local hit ratios than HSCC. On the contrary, HSCC has better global hit ratios overall, because it handles evictions at the homes as well and saves blocks trying to even out the loads of the cluster nodes.

For heavy workloads (the last three, for instance), the local cache hits become less important when compared to the global cache hits. In this case, HSCC outperforms HDC by far exactly because it maintains a higher global hit ratio. As it can be seen from Figure 4.5, saving too many evicted blocks (like HDC does) under memory pressure turns out to be ineffective, as both the local and the global cache hit ratios seem to diminish at the same pace. Thus, the eviction handling must be made with care in order to balance the loads of the caches.

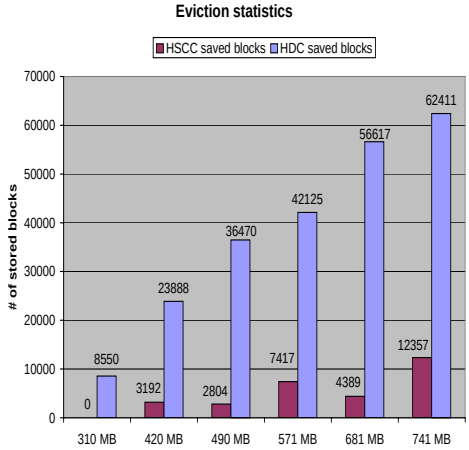


Figure 4.6: Eviction statistics. The number of evicted blocks saved by the CARD driver for each policy

4.7.5 CARD Speedup/Slowdown

We ran the workloads on a CARD driver acting as a remote disk interface (i.e., without enabling the cooperative caching) and we measured the running time using the Unix *time* command. We also ran the workloads on CARD drivers with cooperative caching enabled, both on cold and warm caches. The results are presented in Figure 4.7.

For HSCC, the best speedup was that of the first workload (310 MB). The CARD driver running on a warm cache achieved a speedup of 1.54 over the remote disk interface. Even the workload larger than the global cache (741 MB) experienced speedup, although smaller (see the last line of the x-axis in Figure 4.7). The slowdown of a CARD driver running on a cold cooperative cache is negligible when compared to a remote disk interface (see the middle line of the x-axis in Figure 4.7).

For HDC, practically only the first workload (310 MB) exhibited speedup since that of the second workload (420 MB) is negligible. All the other workloads experienced only slowdowns, both when running the CARD drivers on a cold cache and on a warm one. Moreover, the slowdowns of the CARD driver running on the

cold cache are less severe than those of the CARD driver running on a warmed up cache. Notice however that for large loads (the last three), the HDC slowdowns of a CARD driver running on a cold cache are better than the corresponding ones of HSCC because HDC is a simpler policy than HSCC. Nevertheless, the warm cache figures show that trying aggressively to save evicted blocks is not only a waste, but induces also running time penalties.

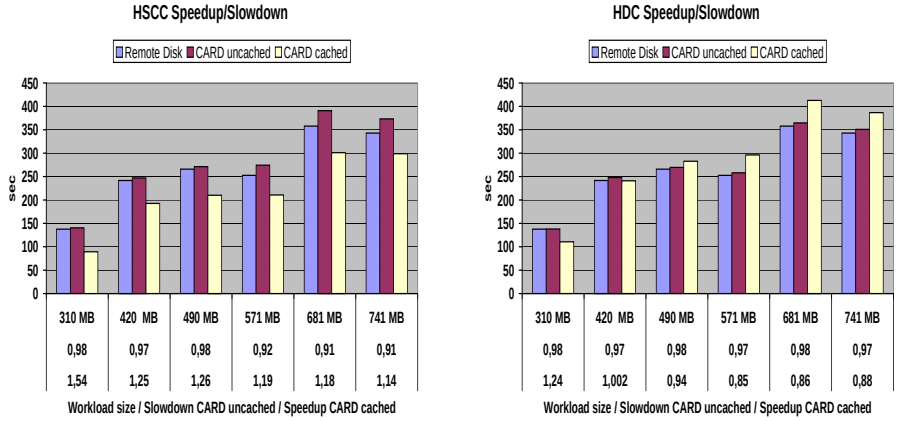


Figure 4.7: Speedup/Slowdown. The cooperative caching enabled operation of CARDs vs. CARDs as remote disk interfaces

Some of the performance numbers of the CARD drivers operating as remote disk interfaces look weird. The 490 MB load takes more time than the 571 MB one. Similar for the (681 MB, 741 MB) pair. This result is equally true when running the workload on the local disk. Therefore, the problem is not related to the CARD driver. Both the 490 MB and 681 MB loads include */usr/share* which is broader and deeper than the other workload directories (*/usr/bin*, */usr/lib*, */usr/src*). The running time breakdowns show indeed that */usr/share* needs more time per MB than the other directories.

4.8 Summary

This chapter presented a flexible solution for a cluster-wide cooperative caching system using Cluster-Aware Remote Disks. A collection of CARD drivers can employ a common cooperative caching policy in order to globally manage the content of the nodes' buffer/page caches. Using a globally managed cache is an easy and natural way to extend cluster-wide the local page/buffer cache while maintaining flexibility through the separation between the data access mechanism (the CARD driver) and the data management policy (the cooperative caching algorithm). Due to this flexibility, we experimented with two cooperative caching policies, Hash

Distributed Caching and Home-based Server-less Cooperative Caching. We designed the latter as a decentralized algorithm that should best match the requirements of parallel processing environments like those of the clusters by evenly distributing the meta-data management among the cluster nodes.

Decentralized algorithms are important for the cluster scalability, but, unfortunately, very few such algorithms have been designed and evaluated. Both HSCC and HDC use decentralized block lookup procedures, but HSCC uses a dynamic placement of the data blocks. Only the meta-data management is statically distributed. HDC uses the block homes both as meta-data managers and as block repositories. Moreover, the block eviction in HDC is done statically, every locally evicted block is always sent to its home. On the contrary, HSCC sends the locally evicted block to other nodes in the cluster based on dynamic knowledge about the cluster load. Our experimental results favor HSCC's approach.

Another decentralized cooperative caching algorithm similar to HDC, PACA [21], uses cluster-wide only one cached copy of any given block. The use of a single cluster-wide copy simplifies the problem of maintaining the consistency across the cluster, but lacks locality of data, an important issue in locality-aware request distribution systems for cluster-based servers. Therefore, we chose HDC to experiment with, as we were further interested in developing Web-caching systems for cluster-based servers.

Our results show that cooperative caching reduces the I/O activity and improves the read latency. Even for heavy workloads (e.g., those larger than the cooperative cache), our algorithm (HSCC) achieves cache hit ratios above 50% without any slowdown (when running on a warm cooperative cache). The slowdowns of our algorithm when running on a cold cooperative cache with respect to the non-cooperative caching performance of the CARD drivers were negligible. The best speed-up observed was 1.54.

There are also some important design issues regarding cooperative caching that are validated by means of a real implementation of the algorithms. First, for heavy loads (i.e., those approaching the size of the global cooperative cache or larger), the local hits become less important than the global ones. Second, saving the locally evicted blocks in remote memories irrespective of the loads of the cluster nodes is a waste and induces running time penalties. For instance, an aggressive eviction handling policy like that of HDC may end up forcing the performance to plummet below that of the same system working on a cold cooperative cache.

The above observations show that a simple and static decentralized scheme like that used by HDC doesn't suit the purposes of scalability. Nevertheless, even if our HSCC algorithm shows better performance, its scalability remains to be proven experimentally. Section 4.5 discusses some theoretical aspects regarding the scalability of the HSCC algorithm.

Chapter 5

Cooperative caching and the cluster-based Web servers

As the previous chapter described the way the CARD drivers use cooperative caching to build a globally coordinated cache by extending the local caching capacity of a disk to the aggregate sum of all the memories in the cluster, one can get now a rough idea of what a SSI cluster-based server using cooperative caching may look like. A server like that in Figure 1.2 services requests that hit a given back-end machine either by fetching the requested document from the global cooperative cache, provided that a globally cached copy of it exists, or by loading it from the disk shared through the CARD drivers by all the nodes in the cluster. The single system image of the server derives from two particular views: that of the user who sees a single generic server endpoint (see Chapter 2) and that of the application server developers who don't need to take into account the distributed nature of the environment because the underlying storage system takes care of it. In other words, the client connects to a server (IP address, server port) pair as if the cluster-based server would be a single server machine, while the sequential application server programs running on the back-end nodes access the data as if stored on local disks. This solution allows using unmodified stand-alone server software on top of COTS clusters.

This image is appealing because its simplicity but hides serious questions regarding the context in which such a server might be used and the way it may affect the effectiveness of the cooperative caching itself. Depending on the environment the cluster-based server is working on, its storage system faces various challenges. For instance, when integrated in multi-tier server architectures using proxies, content delivery networks, etc., the cluster-based servers are responsible to deliver mostly unpopular static documents and dynamic content, as studies [54] show that

the popular document requests are filtered out from the request stream at early stages. The so called “heavy tail” of the request distribution curve (representing large files whose service times account for a considerable part of the total servicing time) remains to be handled at the level of the cluster-based server. As a direct consequence, there is an increased pressure on the back-end level storage system which calls for locality-aware request dispatching [52, 17] or improved caching of the non-popular documents and mitigates the need for load balancing. But the cooperative caching policies are general-purpose algorithms (and so is our HSCC algorithm so far) and they apply an equal treatment to all the cooperatively cached files.

Moreover, cooperative caching has been designed and developed for distributed or parallel file systems and their usual workloads. However, it is a known fact that the Web workloads exhibit particular features both in terms of the access patterns and the size of the requested documents (files), features that differ significantly from those of the regular file system workloads. Namely, the Web requests target mostly the small and popular files that account for a significant part of the total number of requests while only a small fraction of the requests concerns the large and unpopular files. Therefore, cooperative caching needs to be revisited in this context.

The insight gained through the experiments described in the previous chapter adds consistency to the above mentioned concerns. Indeed, we have seen so far that, in distributed environments, certain features of cooperative caching should be handled carefully. For instance, Section 4.7 shows that handling the block eviction irrespective of loads of the cluster nodes may hurt the performance. But a cluster-based server functions like a parallel machine with all the nodes more or less equally balanced (according to some load balancing policy). Under these circumstances, one may consider the eviction handling useless.

If the capability to respond adequately to a given type of workload should be an important factor in developing Web-oriented cooperative caching algorithms, it is equally true that, performance-wise, they should not do worse than simple solutions like document replication. Locality-aware request dispatching algorithms assume most of the time that the documents are fully-replicated across the cluster. The replication causes not only a waste of storage space, but raises also serious administration and maintenance problems as the size of the cluster grows. In fact, nowadays clusters are split between a data and a processing center. The data center gathers the entire storage capacity of the cluster, each of its disks being remotely mounted on disk-less machines from the processing center. However, as concluded by the experiments presented in Section 3.11, virtualizing the remote disks hinders the I/O performance and asks for increased caching capabilities since uncached concurrent non-overlapping block requests have to be serialized at the disk con-

troller. In a system using replicated documents (i.e., the same data set replicated on local disks), such requests take advantage of the disk controller level parallelism. We saw cooperative caching improving the performance of our micro-benchmark running in a distributed environment, but we need to prove that it can also bridge the gap between the performance of the virtual disks and that of the solutions using replicated documents for cluster-based Web servers.

Our response to all these concerns is to steer the cooperative caching operation through hints induced by the application characteristics. In the particular case of the cluster-based Web servers, those hints are deduced by speculating on the Zipf-like [77] Web document request distribution curve in order to cache classes of documents preferentially over other classes of documents. For instance, we show that confining the cooperative caching to the class of the large and unpopular Web documents improves the performance of the general-purpose cooperative caching algorithms. Furthermore, handling the block eviction in a cooperative Web cache pays off only for the significantly large and unpopular documents. To the best of our knowledge, this work is the first attempt to analyze the impact of the application-aware (request distribution aware, in fact) cooperative caching.

Moreover, we supply cooperative caching with support for a fine-grain control over which disk block copies should be kept around in response to a remote request by using the exclusive caching feature of our CARD driver. In this context, we investigate to which extent the exclusive caching can help cooperative caching by avoiding unnecessary copies of certain classes of documents. To the best of our knowledge, this work is the first attempt to combine the exclusive caching with cooperative caching in a cluster-wide, application-aware caching system.

The results presented in this chapter appeared in [50].

5.1 Related work

We presented in Chapters 2 and 3, Sections 2.1 and 3.1, respectively, the background on locally distributed Web servers and remote I/O. At this point, we would like to present past results that attempted to bridge the two areas, as well as the Web workload characterization.

As mentioned in Section 2.1, Locality Aware Request Distribution (LARD) systems [52] attempt to reconcile the locality of the requested data with the load balancing. The paper that first discussed the issue [52] mentioned also that cooperative caching (more precisely GMS [29]) has been tested as an alternative to the LARD techniques, but the results have been shown to be similar. Notice that GMS uses general-purpose algorithms and therefore differs from our approach that tends to fulfill closer the expectations of the applications run on the cluster-based servers (the server programs, in fact) by taking into account their particular features.

An intuition similar to ours led to the results reported in a content-aware cooperative caching system for cluster-based servers developed by Ahn et al. [2]. This system uses a cache replacement policy that strives to avoid document duplicates in the cluster in the following way. As soon as a back-end becomes overloaded, the front-end redirects the requests for a given document to another back-end. The overloaded back-end flushes its own copy of the document as it is clear that it won't be useful locally anymore. Notice that this approach is different than that of our CARD system in which we do not intervene in the operation of the local memory management algorithm. Another difference is that the content-aware cooperative caching algorithm relies on the front-end to provide the block (or rather document) lookup information to the back-end servers. This choice offloads the back-end servers, but has the disadvantages of the centralized algorithms and raises additional questions concerning the scalability of the system.

Both aforementioned systems used simulations to validate their conclusions.

5.1.1 Web workload characterization

Many studies [15, 33, 71] have shown that the Web requests follow a Zipf-like [77] distribution. According to it, the *i-th* most popular document is requested with a probability proportional to $1 / i^{\alpha}$, for $0 < \alpha < 2$. Because of the α values, this distribution is also called *heavy-tailed*. In terms of Web requests, that means the distribution has a long tail of less popular documents with poor locality of reference. The higher the α, the greater the concentration of popular documents.

As a consequence of this distribution, caching popular static documents is very effective, but the higher the fraction of the unpopular requested documents, the lower the caching effectiveness. Moreover, another consequence of this distribution points out the conflict between the load balancing and the data reference locality. The popular documents tend to be served by the same server in order to maximize the cache hit ratios, but that makes the server handling them a hot-spot of the system. Trying to balance the load implies to distribute the service of the popular documents, but doing so creates many replicas of the same document. The existence of multiple replicas reduces the memory available for the service of the rest of the requested documents and entails a poor I/O performance due to the reduced caching effectiveness.

Another consequence, less obvious, advocates for favoring the short-lived connections over the long ones. Crovella et al. [22] showed that, under a Shortest-Connection-First scheduling algorithm, the mean response time improves significantly without affecting the response time of the long running connections too much. Using queuing theory arguments, they proved that this fact is happening exactly because the Web document requests follow a Zipf-like distribution. An ex-

ponential distribution, for instance, would have a more negative impact on the long connections. The result came though at the cost of a significant drop in throughput, mostly due to an user-level implementation. In subsequent work, Harchol-Balter et al. [34] used an improved kernel implementation and showed even better response time figures at no throughput penalty cost.

Another important aspect of today's Web, namely the extensive use of proxies and, more recently, Content Delivery Networks (CDNs), led to another interesting result. These systems deliver replicas of popular documents from points closer to the client than the original server of the documents and thus improve the client-perceived latency, lower both the bandwidth consumption and the load on the original server. As a result, there is a hierarchy of caches that filters out the popular documents of the request stream so that only the less popular documents and the dynamic (uncacheable) content are served by the original servers. Gadde et al. [54] evidenced the phenomenon and called it the *trickle-down* effect. Some of the consequences of this effect are important for our work. First, since the proxies and the CDNs absorb most of the locality of the document stream, having to handle the heavy tail of the stream at the original server exercises an increased pressure on the server storage system. In this context, cooperative caching may play an important role. Second, since only the unpopular part of the request stream hits the original server, the locality-aware request distribution schemes at the original server become important. As an aside, since the source of the load imbalance, namely the popular static documents, has been filtered out, and the locality-aware request distribution is desirable anyway, the importance of the dynamic load balancing at the original server diminishes.

5.2 Caching on a curve

By looking at the Web request distribution curve it is possible to identify entire classes of static documents according to their popularity. For instance, WebStone [61], a commercial benchmark for Web servers respecting a Zipf-like law, identifies by default four classes of static documents: files smaller than 1 KB, files in the (1 KB, 10 KB) and (10 KB, 100 KB) ranges and files larger than 100 KB, denominated by class0, class1, class2 and class3 respectively. In terms of their popularity, class0 accounts for 35% of the requests, class1 for 50%, class2 for 14% and class3 for 1%. However, the servicing time for class3 accounts for roughly one quarter of the total time, while servicing class2 comes close to 40% of the total time.

A legitimate question is to ask whether a special treatment for each class could improve the overall server performance. As seen in Subsection 5.1.1, the results from connection scheduling in stand-alone servers [22, 34] suggest that favoring the short-lived connections in a Shortest Remaining Processing Time connection

scheduling improves the response time without affecting the overall behavior of the server. This result can be an encouraging starting point for our work because it relied on beforehand knowledge of the life length of a connection which was shown to vary consistently with the size of the documents. Put differently, if one singles out the large files as a separate class and schedules connections according to this simple classification, the overall performance of the stand-alone server improves.

By further refining the document taxonomy, we are using a combination of cooperative caching and exclusive caching to manage the global, cluster-wide cache according to the characteristics of the workload. Such an approach needs to respond several challenges. First off, it must be assessed whether cooperative caching compares favorably to simple solutions like document replication. Second, it is unclear whether general-purpose cooperative caching algorithms wouldn't do equally well. Finally, since the previous attempts to use the general-purpose cooperative caching for distributed Web servers [52, 2] relied on simulation, it is interesting to validate our solution through a real implementation.

5.2.1 Cooperative caching

Two of the main features of cooperative caching are of particular interest to our solution. First, the client block requests missing in the local cache are checked against the remote client caches as well before going to the server. Second, cooperative caching implements a global replacement policy for the locally evicted blocks. Due to the aforementioned flexibility of our CARD drivers, we can implement various lookup and eviction handling procedures which allow us testing common sense intuitions about caching documents in the global cache. For instance, the global caching of highly popular documents seems a good idea because it may yield high hit ratios. Also, saving evicted blocks for unpopular documents seems equally important because they account for a significant part of the total servicing time. Since all the classes compete for the same memory, it is questionable whether a general purpose cooperative caching algorithm would do well. If the answer is negative, one needs to find out whether policies trading off among classes of documents wouldn't do better.

As pointed out in Subsection 5.1.1, the servers operating in complex environments using proxies, content delivery networks, etc. serve mostly the unpopular documents as the popular ones are filtered out of the request stream at early stages [54]. This is yet another argument to assess the performance of caching exclusively the class of the unpopular documents.

In Subsections 4.7.3 and 4.7.4, we saw that handling the block evictions irrespective of the loads of the clients participating in the global cache may cause a severe performance loss (the performance can plummet below that of a cold global

cache). However, these results are valid for cooperative caching in a distributed environment. A natural question asks whether the global eviction handling is a right idea in a cluster-based Web server, since such a server can be considered a parallel machine in which all nodes are (more or less) equally busy (especially when load balancing mechanisms are used). That is to say, the chances to find lightly loaded nodes to host the remotely evicted blocks are small and, in such circumstances, our previous experience argues against the global eviction handling.

One final question asks whether cooperative caching can compare to a simple technique like replication. If not, one loses an important advantage of replication, namely that of the disk controller parallelism. In the worst case, when all the requests have to go to the disk, cooperative caching pays double: the protocol overhead and the sequential disk processing, the latter being probably the heaviest price. And if this is the case, it is interesting to assess whether mixing cooperative caching with replication (that is, replicating some classes of documents while cooperatively caching the other) alleviates these consequences.

We respond to all these questions by writing specific algorithms that all build upon the HSCC algorithm. By specific algorithms we actually mean various versions of the lookup and eviction procedures of HSCC, written according to the definition in Section 4.3. For instance, deciding not to cooperatively cache a certain class of static documents is simply a matter of downloading into the CARD driver a lookup procedure that lets the requests addressed to the files from that class to be directly forwarded to the disk node. Similarly, handling the evictions of the blocks belonging to the files in a certain class of documents requires downloading into the CARD driver an eviction handler that tries to find a remote host for the locally evicted blocks of the files in that class. For the case when no eviction is to be handled, a null eviction procedure is downloaded into the kernel in order to disregard the block eviction events.

5.2.2 Exclusive caching

As seen in Section 3.7, our exclusive caching solution operates cluster-wide by avoiding double buffering as a host mounting a remote disk through a CARD driver requests remote blocks. When used with cooperative caching, the exclusive caching concerns only the requests that need to go to the disk. The requests serviced from the remote client caches don't need to worry about remote copies, because they got there according to the joint management algorithm of the global cache (loaded on demand or saved as a result of remote evictions), while the eviction from that cache is regulated through the global replacement policy.

The exclusive caching enables the computer-attached disks in COTS clusters to exhibit also a network-attached behavior, provided that no other useful compu-

tation takes place on that node. A disk with a network-attached behavior mounted remotely through CARD drivers opens also possibilities for selective caching according to certain criteria. For instance, in terms of Web workloads, the page/buffer cache at the disk node can be used to store unpopular documents and thus can act as a "discard cache" for unpopular files. In general, setting up a "discard cache" for a given class of documents at the disk node is simply a matter of suppressing the exclusive caching flag in the requests for the uncached blocks of the files belonging to that class. As a result, the disk node cache and its *local* replacement policy govern the caching of that class of documents for which, presumably, it may make little sense to cache its documents across the cluster. Naturally, the effectiveness of such a method depends on the memory size of the disk node and the size of the class.

5.3 Performance evaluation

In order to evaluate the performance of a cooperative cache enabled SSI cluster-based Web server, we use WebStone [61], a well known commercial benchmark for Web servers respecting a Zipf-like [77] document retrieval distribution. We instruct the WebStone software to retrieve static documents only. Therefore, throughout the rest of this section, by WebStone operations we refer to HTTP GET commands. The benchmark uses HTTP 1.0 and a workload around 1 GB of data (corresponding to a figure of 300 simultaneous connections maintained by the client to the server). The server uses Linux *ext2* file systems both on top of local disks and remotely mounted disks. The CARD drivers have the exclusive caching turned on by default (if not otherwise stated).

5.3.1 Experimental setup

We run our experiments on a 3-node Linux cluster using the setup described in Subsection 3.10.1. For these experiments, we use an upgraded version of the Myrinet GM driver, namely GM 1.6.4 [66]. Only partitions of the IBM DCAS-34330W Fast/Ultra-SE SCSI disks are mounted remotely for the experiments that we further describe. The disks are formatted with a native Linux file system format (*ext2*). The file systems are non-aged (0.7% non-contiguous, as reported by the *fsck* command). The server machines mount the remote disk partitions read-only.

As a Web server we use Apache 1.3.20 [5]. A Linux router stays between the client and the server machines. Both the client and the router are Athlon AMD XP 1.5 GHz PCs with 512 MB of RAM and run Linux 2.4.19. The client, the router and the server(s) are all interconnected through regular 100Mb/s Ethernet. Figure 5.1 describes visually the experimental setup.

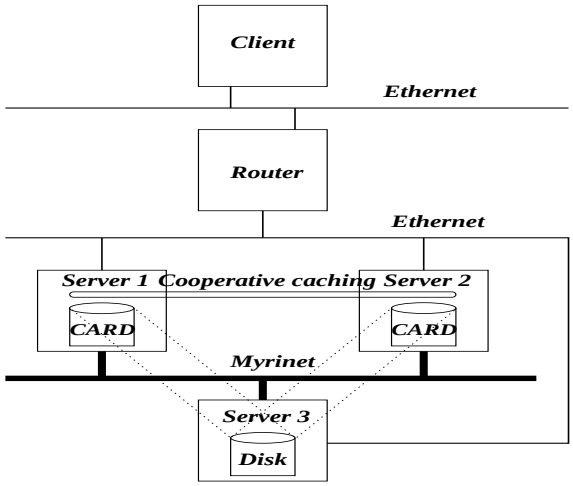


Figure 5.1: Experimental setup for request distribution-aware caching

5.3.2 Experimental methodology

The WebStone benchmark runs on the client machine and sends requests to the cluster-based server through the router. The role of the router is twofold: to induce additional latency in order to emulate some Internet-like behavior (almost impossible to be noticed on a LAN) and to dispatch the client requests according to a Round Robin policy to two of the server machines in order to yield perfect load balancing (from the perspective of the number of the serviced requests). The two cluster nodes build a cooperative cache of size at most 512 MB (somewhat smaller, in fact, due to the space occupied mainly by the operating system and the server program). This value amounts to at most half of the aggregate storage size of the working set of the workload (1 GB of data). We chose this value in order to avoid two situations: the case when the working set fits in any of the local memories, as well as the case when the workload fits entirely in the global cache. The two nodes mount the disk containing the benchmark file set through CARD drivers. The disk itself resides on the third server node. The page/buffer cache at the disk node (the third server machine) is referred throughout the rest of the section as the “discard cache”.

5.3.3 Preliminary discussion

In Subsection 3.11.4 we described an experiment assessing the performance of the CARD drivers acting as simple remote disk interfaces, i.e., without cooperative caching. The experimental results (see Figure 3.9) pointed out the performance

loss of a cluster-based server using disks mounted remotely by means of CARD drivers when compared to a fully-replicated solution (that is, the servers have all the needed documents stored on local disks). When equipped with local disks, the servers performed best because the load was almost equally split between the two disk drives that we used and that maximized the amount of the disk parallelism. We take those experimental results as a reference for all the graphs depicting the results of the experiments that are subsequently described. The next subsections attempt to quantify to which extent the request distribution-aware cooperative caching can make up for that performance loss and, very important, under which circumstances.

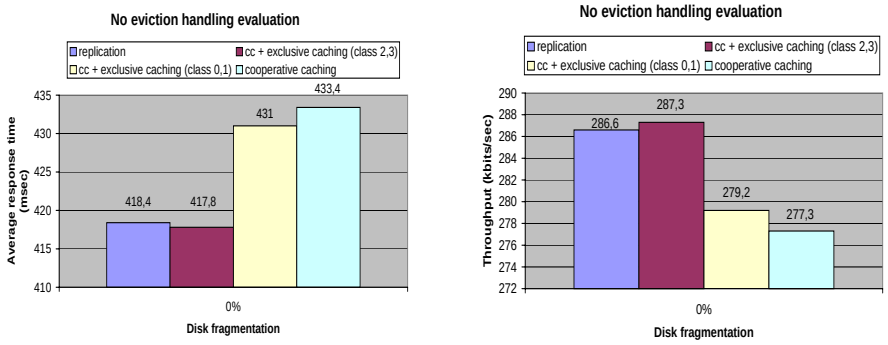


Figure 5.2: WebStone evaluation of cooperative caching without eviction handling

5.3.4 The performance of cooperative caching without eviction handling

In a first experiment we assess the impact of looking up cached copies of a locally missing block in remote client memories without considering any global replacement policy (i.e., without eviction handling). In order to do that, we wrote three policies and we compared their performance to that of the fully-replicated solution. Using the class definitions in Section 5.2, a brief description of these policies follows:

- cooperatively cache class2 and class3 files, that is, the files that require most of the servicing time. The small and popular documents are cached at the discard cache
- cooperatively cache class0 and class1 files (small and popular documents) and rely on the discard cache to cope with class2 and class3 documents
- cooperatively cache all the requested documents

The results are presented in Figure 5.2. Notice that the first policy has the best results (even slightly better than the fully-replicated solution), which emphasizes the importance of caching the heavy tail of the request distribution curve. The last two policies exhibit performances comparable to each other (with a slight degradation for the results of the plain cooperative caching). This outcome can be explained if we remember that class0 and class1 account for 85% of the requests. Thus, using the limited capacity of the discard cache to store the large files of class2 and class3 doesn't improve significantly the performance.

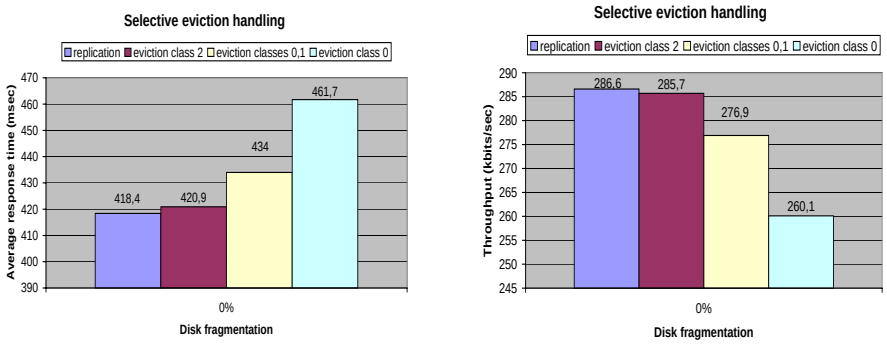


Figure 5.3: WebStone evaluation of selective block eviction handling according to classes of documents

5.3.5 Selective eviction handling

We pushed our investigation further by attempting to assess to which extent saving locally evicted blocks in remote client memories affects the performance of the cooperative caching operating in a cluster-based server environment. In our evaluation, we used three policies that handle evictions selectively, for given classes of documents only. Using again the notations from Section 5.2, the definitions of these policies are:

- handle only the evictions of the blocks belonging to the files of class2, those that account for almost 40% of the total servicing time
- handle the evictions of the blocks of the class0 and class1, that is, popular documents less than 10 KB
- handle the evictions for the class0 only, i.e., popular documents, representing 35% of the requested files

The results are reported in Figure 5.3 (which also compares them with those of the fully-replicated solution). Notice that the first policy comes very close to the

performance of the replicated solution which underlines again the importance of keeping the class2 files cached in memory, this time due to the eviction handling. The performance degradation of the other two policies shows that the smaller the file, the less important the block eviction handling. As the sizes of the documents grow, the eviction handling becomes more important. This effect can be noticed by looking at Figure 5.3 from right to left.

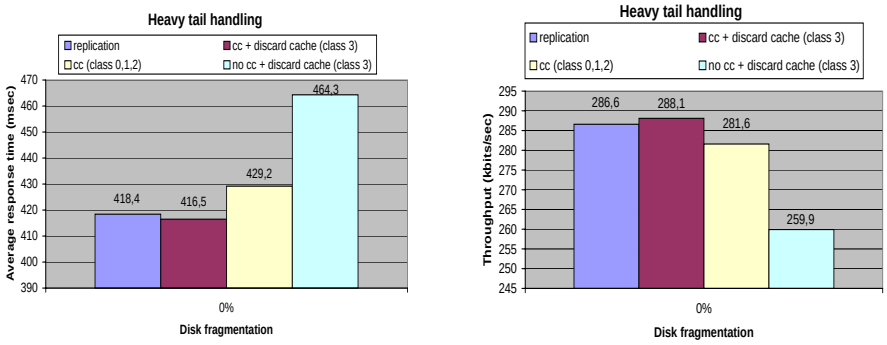


Figure 5.4: WebStone evaluation of heavy tail caching

5.3.6 Handling the heavy tail of the request distribution curve

The heavy tail of the request distribution curve represents 1% of the requested documents but takes some 25% of the total servicing time. Our previous experiments showed the importance of caching the large documents of class2 and class3. In this subsection we try to get more insight about this issue by separately treating the highly unpopular documents. We also attempt to simulate the operation of the multi-tier servers that filter out the request stream by letting only the heavy tail to be served at the original server. The two servers do that by cooperatively caching all the classes of documents but class3, which remains to be stored at the discard cache. We wrote three policies using cooperative caching that handle evictions in class2:

- cooperatively cache all the documents and use the discard cache to keep copies of the large documents of class3
- cooperatively cache class0, class1 and class2 documents without caching class3 documents at all
- cache only class3 documents using the discard cache

The results are shown in Figure 5.4 and, as usual, they are compared to those of the replicated solution. The poor performance of the third policy shows that

caching the large files doesn't help if all the other requests go to the disk. This conclusion becomes clear when comparing the performance of the third policy with that of the second policy which doesn't cache class3 documents at all and yet performs significantly better. The best solution is offered by the first policy which shows a slightly better performance than the solution using replication.

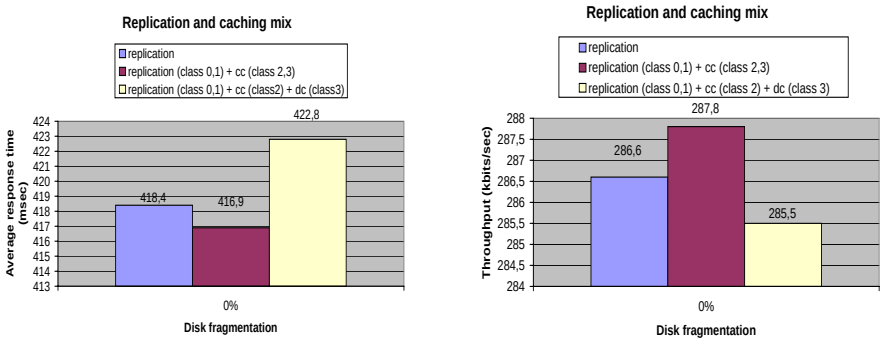


Figure 5.5: WebStone evaluation of combining replication with caching

5.3.7 Mixing replication with cooperative caching

In this subsection we present the results of a mixed solution that uses both replication and cooperative caching (see Figure 5.5). We wrote and tested two policies. In the first one, we replicate the class0 and class1 files on the local disks of the two servers and use cooperative caching for the class2 and class3 documents. The second policy uses replication for the small and popular documents (class0 and class1 files), cooperatively caches the class2 files and uses the discard cache to store the unpopular documents (class3 files). Both policies handle evictions for the class2 files only. The first policy outperforms the fully-replicated solution and its results are consistent with our previous observation according to which caching class2 and class3 files yields the best performance. The fact that using the replication for the small and popular files doesn't affect this conclusion indicates that these classes enjoy enough locality due to their popularity (since there are only two servers, there is a 50-50 probability that a second request for the same document will hit the same server).

5.4 Summary

In this chapter we described a request distribution-aware caching system for cluster-based Web servers. Using cooperative caching and exclusive caching as driving engines, our caching system speculates on the properties of the Zipf-like request

distribution curves for static Web documents by selectively caching classes of documents according to their popularity. We also took the opportunity to investigate the effects of such a particular type of caching on a general purpose technique like cooperative caching.

The experimental results gathered by running a well-known commercial Web benchmark, WebStone, help us reach some conclusions. First, cooperative caching is a useful technique for the SSI cluster-based Web servers (especially when targeting the class of the large and unpopular files) because it can bridge the performance gap between the solutions serving documents through virtual disks and the fully-replicated solutions. Similarly, handling the block evictions pays off only for the unpopular and significantly large documents. Attempting to handle the block evictions for the small and popular documents penalizes the performance of the system. A separate handling of the heavy tail of the request distribution curve may bring further benefits if the previous observations are taken into account.

Chapter 6

TCP connection endpoint migration

One of the main features of our SSI cluster-based server, as presented in Chapter 2, is the back-end level, policy-oriented request distribution mechanism. When a request hits a back-end server, a policy downloaded in the kernel by the application server has to decide where the request should be handled. A possible outcome is to hand in the request to another back-end server, provided that such a decision yields better performance. Simple decisions can be reached independently of the request content by considering the various loads of the cluster nodes. Some other decisions, however, need to spy on the request in order to gather information that allows reaching a certain quality of request service. Typical such decisions are those performing content- (or locality-) aware request routing. However, this type of information becomes available only after a connection has been already set up between the client and one of the back-end servers. As soon as that happened, a request routing mechanism has to deal one way or another with the back-end that was involved in the connection setup, since choosing a new server to service the request leaves behind a connected server-side endpoint. Most of the solutions to the problem (see Section 2.1) choose to circumvent the issue by using a switch interposed between the client and the cluster-based server. The switch reaches request routing decisions early, before the connections to the server machine have been set up. Alternatively, other solutions let the client connect to a server which becomes a relay for all the messages exchanged between the client and the cluster-based server as soon as the request has been routed to another back-end server.

We present in this chapter a mechanism that deals cleaner with the aforementioned problem by migrating dynamically server-side TCP connection endpoints between the back-end servers. The mechanism suits very well the SSI design of our

cluster-based server, since it establishes a functional equivalence of all the server nodes in the cluster. As soon as a client establishes a connection to an arbitrary server, any further server-side endpoint migration remains undetected on the client side, without affecting in any way the correctness of the request service.

Thus, we affirm that the TCP connection endpoint migration is a flexible way of assigning and reassigning server-side connection endpoints to particular back-end machines. It hides from the client the distributed nature of the server, for the client sees only a generic server-side endpoint to which it connects, irrespective of its actual physical server binding. The mechanism has been implemented as a Linux kernel module and complies with the software architecture described in Section 2.3. The TCP connection endpoint migration can be used together with load-balancing algorithms or even content-aware request distribution.

The rest of this chapter begins by discussing the related work. Then, the design and implementation of the TCP connection endpoint migration protocol are presented at large. Finally, we show by means of experimental results that the TCP connection endpoint migration suits well the purposes of the back-end level request distribution algorithms for cluster-based Web servers, both for persistent and non-persistent HTTP connections. The experiments used both academic and commercial benchmarks.

6.1 Related work

Connection migration is a new mechanism that has been only lately put under scrutiny by Snoeren et al. [59, 60] and Sultan et al. [64, 63]. Their solutions describe client-server migration protocols that allow either one of the involved parties a graceful migration of their corresponding endpoint to a third party conforming to the protocol. No front-end or switch is needed between the client and the server. The first solution is not a true migration protocol as it involves an user-level "wedge" that intermediates between the connection endpoints. Moreover, that protocol is application-dependent (i.e., not a TCP-migration protocol).

However, there is little evidence that the client-server connection migration could be successfully used in request distribution for cluster servers mostly because of the incurred overhead. In fact, Snoeren et al. used it for fault-tolerance purposes, as a fine-grain fail-over mechanism for long-running connections switching across a distributed collection of replica servers [60]. In a somewhat different domain, they used connection migration to approach host mobility [59]. With Server Continuations, Sultan et al. [63] use the connection migration to migrate server sessions as a particular form of process/thread migration.

Our protocol is an application-independent, server-side connection endpoint migration protocol (server-to-server, client-transparent). It has versions both for

the architectures employing front-ends and for the fully-distributed ones. Its main advantages are performance-related. Since it is client-transparent, the protocol is not sensitive to the Internet behavior. In the case of the general client-server migration protocols, the inner mechanisms of TCP (slow-start/congestion avoidance, exponential back-off) may negatively affect the migration. The possible performance degradation renders thus questionable the use of connection migration in request distribution policies. On the contrary, a server-side protocol takes advantage of powerful backplane interconnects which have better performance figures than the Internet. A client-transparent migration protocol has the disadvantage of employing an additional connection router (either the front-end or the server node itself). For fully-distributed cluster-based servers (i.e., when no front-ends are used), the problem is mitigated by the low costs of passing messages over SANs.

The client-server migration protocols raise more general questions as they can be used to migrate connections over Wide Area Networks as well. In such a context, a client-aware migration protocol breaks the client-server paradigm: a client is not anymore connected to a single server, represented through an (IP address, TCP port) pair, but rather to a collection of servers identified by a set of tuples with different IP addresses and the same TCP port. While the distributed nature of the server contributes obviously to improved service, making it public does not provide a new abstraction, it just complicates an existing one. While effective, we do not find it to be a neat design. We find Anypoint [74], a one-to-many communication model, a far more consistently and properly designed solution. Anypoint uses also application-layer policies to route the requests and operates at the granularity of transport frames. Anypoint's performance has been tested for an NFS storage router and not for the Web.

Also, in some sense, a client-sever migration protocol is not very different from the existing HTTP-redirection protocol (see Section 2.1), namely, they both involve the client in redirecting the connection, although in the HTTP-redirection case the redirection is done by setting up a brand new connection. While technically different, both protocols suffer from the performance penalty of having to go over the Internet to perform the redirection.

A client-transparent protocol can be regarded as a particular client-server migration protocol involving a server and a client stand-in. This representative of the client is either the front-end (when taking part in the request routing) or one of the servers, as we will see for the case of the fully-distributed request routing. Since this representative is directly linked to its peer (the server representing the migration target), the migration protocol is faster. Moreover, the client-server paradigm remains unaffected if all the servers in the cluster use the same virtual IP address. The client communicates with a locally-distributed server as if it would do with a stand-alone one.

6.2 Background

This section aims at presenting briefly those features of the TCP protocol and its typical kernel implementation that facilitate a better understanding of our TCP connection endpoint migration mechanism. To do so, we adopt two perspectives: that of the system programmer who writes client-server applications using TCP/IP, and that of the kernel developer who provides the application developer with the right operating system services (system calls). Our main references for the TCP protocol are RFC 793 and 1072 [41, 42] and Stevens [62]. For the kernel implementation of the TCP protocol we assume the Linux case.

6.2.1 The perspective of the application developer

An application developer needs system support for establishing a connection between its client and server applications, for exchanging messages between the two and for shutting down the communication between them. In order to simplify the explanation, we rely on a widely used library of network services, the BSD socket library. While there are other libraries supporting similar services, we believe the BSD socket library to be illustrative enough for our purposes.

Before communicating with a server over TCP/IP, a client has to setup a connection with the server by using the *connect* system call. On the server side, the server program uses an *accept* system call to wait for incoming connection requests. As soon as the *rendez-vous* synchronization takes place and the connection between the client and the server is set up, the server program unblocks from the *accept* system call and the useful communication can take place on the freshly setup connection. Sending and receiving messages can be accomplished through a variety of methods including the regular *read/write* system calls for file systems.

As soon as one of the peers wants to tear down the connection, it uses the *close* system call. There is also the possibility to close partially an endpoint of the connection by calling the *shutdown* system call with the appropriate parameters. Basically, *shutdown* allows closing either the sending or the receiving part of the local connection endpoint, or both (in this case being the equivalent of *close*).

6.2.2 Processing the network traffic in the kernel

The modern operating system kernels use an event-driven model to process the network traffic. The network card interrupt handlers store packets in a general purpose queue and schedule the appropriate software interrupts (*bottom halfs* in Linux) to handle the queue. The protocol processing runs in these software interrupt handlers that pass the processed packet from the general purpose queue to special purpose

queues managed by the protocols the packets are intended for. A TCP/IP packet, for instance, is passed from the general purpose queue to the IP software interrupt handler, the IP-specific processing is carried out and then the processed packet is delivered to the TCP handler. In turn, this handler executes the TCP-specific protocol and stores the packet in a particular queue (in fact, either the *listen* queue, for connection setup packets, or the socket *receive* queue, for regular packets addressed to an already established connection). This particular queue (either the *listen* or the *receive* queue) will be processed by the targeted process when running in kernel mode as a consequence of executing a system call. In our example, *accept* will process the *listen* queue while one of the *read/readv/recv/recvfrom/recvmsg* will take care of the *receive* queue. This event driven model aims at minimizing the protocol processing overhead, as the targeted process doesn't need to wait actively for incoming packets (polling is a bad idea for non-preemptive kernels like Linux, at least for the versions before 2.6). The incoming packets are placed in a queue at interrupt time and, later on, the rest of the processing takes place when the application (in our case, the server program) runs in kernel context.

6.2.3 The three-way handshake connection setup protocol

The aforementioned rendez-vous between a client calling *connect* and a server waiting in an *accept* will be explained now from the kernel perspective. When a client sends a connection setup request to a server (through a *connect* call), the following steps take place:

- The client TCP engine sends a SYN packet to the remote peer. The SYN packet contains, among other things, an initial sequence number (ISN) that is used to mark the right sequence of packets that will be sent by the client.
- The server receives the SYN packet and responds on the spot with a SYN_ACK packet that contains the server ISN. Also, the SYN_ACK packet acknowledges the received SYN packet by adding one to the client ISN.
- When the server SYN_ACK reaches the client, this one replies with an ACK segment that increments the server ISN by one.

The server kernel is affected by this protocol in the following way. As soon as the server acknowledges the client SYN, an *open-request* structure is added to a SYN_Q queue (or *listen* queue, as defined in the previous subsection) associated with the listening socket. When the client ACK arrives, the corresponding *open-request* at the server is marked "ready" and a freshly created socket is associated with it. Variations of this scheme used in other kernels employ two queues, a

SYN_RCVD and an *accept* queue. The incoming SYNs are placed in the SYN_-RCVD queue and, when the ACK comes, the entry is released and a new socket is placed in the *accept* queue. Regardless of the implementation, later on, when the user application-invoked *accept* runs in kernel context over the SYN_Q (or the *accept* queue, for other systems), the newly established socket is passed on to the server application and the corresponding *open-request* structure is released.

6.3 TCP connection endpoint migration overview

As seen in Section 6.1, our TCP connection endpoint migration is client-transparent and targets locally-distributed server architectures. It has two variants, one involving front-end(s) and a fully-distributed one. As a request (a SYN packet, in fact) arrives at the front-end, it can be directed to a given server according to a certain policy or it can pass through to hit eventually a back-end server.

In the first case, the front-ends keep a mapping table holding (connection ID, server ID) entries. Every packet flowing in along the connection will be routed by the front-end according to this mapping table. If a back-end server chooses at some point to migrate its connection endpoint to another back-end server, the corresponding mapping entry at the front-end has to be updated as well. A single cluster may use many front-ends in order to ensure the scalability of the server. At the back-end level, a context associated with each connection endpoint stores the identity of the front-end through which the connection "came" first. This information helps clean up the mapping table once the TCP connection has been closed.

In the second case, when the requests reach the back-end servers without front-end involvement (at least from the connection routing point of view), the assignment of a connection to a back-end has to be done by an external entity. Examples of such entities include the DNS servers resolving name queries according to a Round Robin algorithm [16] or front-ends that use a static connection assignment according to some hash function on the connection ID (client IP and/or client port). If the chosen back-end decides at a later time to migrate its connection endpoint to another server in the cluster, a (connection ID, server ID) entry will be inserted locally in a so called forwarding table. All the subsequent packets of the migrated connection will then be forwarded according to the corresponding table entry.

Regardless whether the front-ends are involved or not, the connection endpoint migration protocol takes place between two back-end server machines and has two main stages. First, the back-end initiating the migration sets up a connection with the new server as if it would be the client currently using the connection about to be migrated. The back-ends do so by fulfilling a modified version of the connection setup protocol described in Subsection 6.2.3. As a result of this step, a new server-

side connection endpoint intended to be a duplicate of that of the initiating server is set up at the new back-end. Then, in a second step, this new endpoint becomes truly a duplicate of the old server's endpoint when the initiating back-end transfers the entire server-side connection status to the new site. The transfer is made such that, at the end of a successful migration, the client can continue exchanging messages with the new server from the point it had left the communication with the old server. The old endpoint is deallocated and the client-server communication resumes by using the new server-side endpoint. All this happens without the client's knowledge. The next sections describe in detail the two versions of the migration protocol.

6.4 The front-ends and their role in the TCP connection endpoint migration

The uninformed routing performed at the front-end snoops the incoming packets passing through the router and maps their destination address onto a Medium Access Control (MAC) [62] address in order to deliver them to their destination hosts. The mapping is computed according to some hashing function (regardless whether this function is load-aware or not). Such a solution requires that all the back-end servers share the same IP address. Sharing the same IP address is done by setting the *Virtual IP* of the cluster as an *IP alias* on each back-end machine. Some solutions like Network Dispatcher [37] and ONE-IP [24] create an entry in a mapping table at the front-end for every new connection request (i.e., for every new incoming SYN packet). The entry maps the *Virtual IP* to the MAC address provided by the hash. Allocating a new entry each time an incoming SYN packet passes through the front-end raises additional problems if many SYN packets are discarded later on at the back-end level by overloaded servers. Namely, a clean-up mechanism is needed in order to flush the useless/stale entries from the table.

In our approach, the connection-to-MAC mapping relies on a simple hash function that yields the same MAC address for all the packets of a connection. Thus, if no connection endpoint migration occurs, there is no need to store an entry in the mapping table. If later on the server connection endpoint is migrated to another node, an update message sent by the new server triggers the allocation of a new mapping entry. From this point on, all the client packets will be routed according to the newly created entry and the hash function will be disregarded. The entries will be deallocated either when the TCP termination protocol [41, 62] takes place or when either party sends a RST [41, 62] segment. This solution reduces the connection handling overhead as not all the connections end up being migrated. Moreover, there is no need to clean up unnecessarily allocated entries. Also, the

memory consumption is reduced.

It is also worth mentioning that the front-ends act as routers for the incoming packets but not necessarily for the outgoing ones. Only the outgoing packets that need to change the connection-to-MAC table (such as FIN, RST or update packets) are forcibly routed through the front-end they correspond to. All the other outgoing packets can flow out through regular IP routers. This aspect is important when one wishes to dedicate the front-ends to the connection routing job only.

So far, our cluster architecture could have been that of a general *Cluster-based server*. But the use of a *Virtual IP* at the back-ends makes it seem like a *Virtual server* (see Section 2.1) in which all the server machines have to be connected to the same LAN segment so that they can see each other's MAC address.

6.5 Request routing without front-end involvement

The front-ends are not necessary to the connection endpoint migration. By making a back-end act as a "fake front-end", for instance, one gets a fully distributed connection endpoint migration architecture. As long as the front-ends cope decently with the load, the advantages of a fully distributed solution may appear farfetched. Moreover, with the front-end replication, even the scalability issue seems solved.

However, a front-end solution has its disadvantages. First, any migration at the back-end level results in an update message sent to the front-end. This additional message consumes payload (LAN) bandwidth and, more important, CPU processing time to handle the interrupts at the front-end, which is known to be a serious bottleneck for the network systems. Second, as soon as a migration succeeds, all the client packets flowing through the front-end have to be looked up against a hash table. The lookup may be especially cumbersome for the persistent HTTP connections, for instance. For these connections, the subsequent requests passed along the connection will have to pay for the previous (and possibly unrelated) request routing decisions. With a fully distributed solution circumventing the front-ends, such shortcomings are alleviated, provided that some load-balancing scheme dispatches evenly the requests to the "routing" servers at the back-end level. Such a balancing scheme can be as simple as an equivalent of the Round Robin DNS [16].

Making a back-end server also a connection router, as soon as one of its connection endpoints migrates to some other node, is simple enough. As soon as a connection endpoint migrates somewhere else in the cluster, a (connection ID, new server ID) pair is recorded in a so called forwarding table at the old server. Since no trace of the connection endpoint has been left behind at the old server, the subsequent packets for the migrated connection arriving at the back-end are simply delivered to the usual RST mechanism of TCP [62, 41]. The routine responsible to

send back a RST segment to the client has been wrapped up with code that checks first to see whether the forwarding table has an entry for that connection. If so, the packet is not discarded and no RST segment will be generated. Instead, the packet will be sent further to the new server which is identified according to the information stored in the forwarding table. It is also worth adding that wrapping up the kernel routine responsible to send a RST segment is simply a matter of replacing a function pointer in the Linux kernel. When the new server closes the connection, it sends back a cleanup message to the old server over the SAN in order to flush out the corresponding entry from the forwarding table.

6.6 The connection endpoint migration protocol

Once a connection is established between a client and a back-end machine, its server-side endpoint can migrate to another node at will. The operation of a server wishing to migrate one of its connection endpoints to another back-end server is presented by the state machine in Figure 6.1. The MIGRATION_WAIT state is not a new TCP state because the connection remains in the TCP_ESTABLISHED state. But for a better understanding of the protocol, we decided to introduce a wait state describing the operation of the network connection during the migration. As it can be seen, as soon as a migration request is issued (i.e., a migration SYN packet is sent to the remote server), the connection moves to a wait state in which every incoming client packet is stored in a checkpoint associated with the connection.

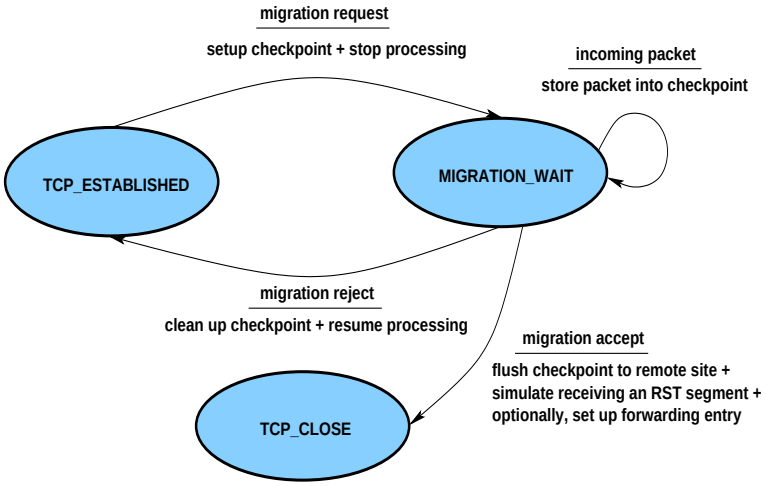


Figure 6.1: Connection migration operation at the initiator

Upon receiving a response from the remote server, the connection leaves the

wait state. If the migration failed, the connection checkpoint is cleaned up by delivering the packets to the connection and the connection processing resumes. On migration success, the checkpoint content is sent to the remote site and the local server-side endpoint is deallocated. Figure 6.2 shows a visual description of the individual protocol steps of a successful connection endpoint migration.

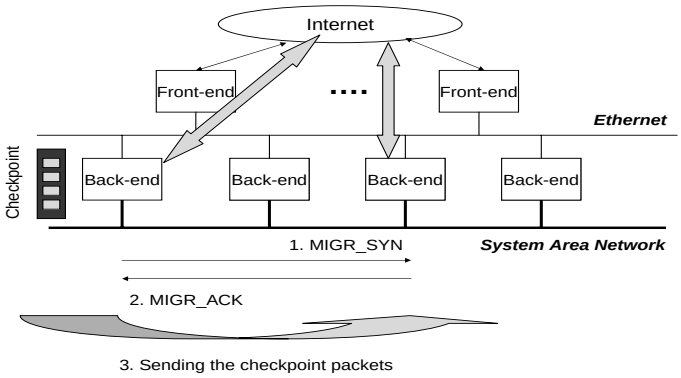


Figure 6.2: Connection endpoint migration at a glance

As already mentioned, the connection between the two back-end servers gets set up such that, after transferring the server-side endpoint status to the new site, the client can continue exchanging messages with the new server as if nothing happened. In order to do so, certain conditions have to be met. Some of them regard the sequence numbers and the time stamps used by the TCP protocol, some other concern the aforementioned checkpoint established in order to cope with the client packets that have been received but not yet serviced during the migration. We discuss these issues in the context of the two main steps of the connection endpoint migration protocol: the server-to-server connection setup protocol (i.e., the modified three-way handshake) and the migration completion phase.

6.6.1 Matching the sequence numbers

Upon receiving a migration SYN, a server targeted by a migration will duplicate locally the endpoint of the initiator of the migration. This operation needs properly negotiated initial sequence numbers (ISN) to be used in the three-way handshake protocol (see Subsection 6.2.3). The packets flowing between the client and the new server have to respect the sequence numbers agreed upon with the old server by the time of the migration. Since the new server-side endpoint is set up by relying on the TCP connection setup protocol, the initiator of the migration has to choose a proper “client” ISN. Additionally, it has also to impose the ISN that the target

server will use as its own ISN during the connection setup protocol described in Subsection 6.2.3.

Enabling the new server to choose an ISN of its own would not permit matching the server sequence numbers currently in use. Namely, the ISN of the new server should be the next allowable send sequence number (denoted by *snd_nxt* in RFC 793 [41]) for the old server. To do so, the initiator server uses a modified SYN segment carrying with it the "imposed" server ISN to be used at the migration site. When the SYN packet is processed at the remote TCP stack, the "imposed" ISN is adopted as a "locally generated" ISN.

The server initiating the migration chooses the "client" ISN to be either the sequence number of the first packet stored in the connection checkpoint or, if that checkpoint is empty, the next sequence number usable by the client (denoted by *rcv_nxt* in RFC 793 [41]), minus one. By choosing the two ISNs in this way, the three-way handshake protocol will ensure that the connection endpoint at the new server site respects the sequence numbers agreed upon between the client and the old server by the time of the migration.

6.6.2 The modified three-way handshake connection setup protocol

A particular feature of this migration connection setup protocol is that the last two messages in the three way handshake do not go over the SAN between the peers. Under this modification, the protocol described in Subsection 6.2.3 becomes:

- The initiator server sends to the new server a modified SYN packet that carries the "client" and the "imposed" ISNs chosen as specified in Subsection 6.6.1. The "imposed" ISN for the new server is adopted at the remote site as a "locally generated" ISN.

- The new server receives the SYN packet, carries out the typical connection setup processing with the supplied "imposed" ISN but drops the generated SYN_ACK packet. Instead, it generates locally the corresponding ACK by modifying the received SYN packet (i.e., by changing accordingly its sequence numbers and by erasing the SYN flag).

As it can be noticed, only one message is necessary to set up a new connection endpoint at the new server site. This design saves us one SAN message (namely the final ACK in the TCP connection setup protocol, since, as we will see in the next subsection, a migration acknowledgment message is needed anyway) and is part of our strategy to keep the migration overhead as low as possible in order to be able to use the connection endpoint migration in request distribution.

6.6.3 Completing the migration

As soon as the new connection endpoint is fully set up at the remote site, the new server sends back a *migration acknowledgment* to the old server (see Figure 6.2). In turn, the requester flushes out the connection checkpoint and the write queue (containing packets sent by the server but not yet acknowledged by the client) by sending them to the new server site. Then, the remote site is able to re-play safely the checkpoint because of the properly set sequence numbers. As for the machine that initiated the migration, it simulates receiving a RST segment (see [41, 62]) in order to flush out the connection state and the allocated resources.

One tricky business is to find out whom to send the *migration acknowledgment* to, since the new server thinks the new endpoint has been set up for yet another client connection. Essentially, the TCP connection setup protocol provides no means to distinguish between a connection setup request and a migration request (since, obviously, it has not been developed to support migration). The solution is to piggyback the ID of the old server on the migration SYN message by using the *urgent pointer* field in the TCP header [41, 62]. This choice may not seem neat but is truly harmless as the urgent pointer field of the SYN segments has no particular meaning. At the new server node, the ID will be stored in the environment of the newly created socket and, when the time for the *migration acknowledgment* comes, it will be used to direct the message to the initiator of the migration. As a technical comment, it is worth saying that this value is not stored in the open-request structure mentioned in Section 6.2, but saved from the migration SYN packet into the urgent pointer field of the locally generated ACK. From here, it will be saved in the newly created socket data structure as the ACK segment gets processed.

6.6.4 The role of the connection checkpoint

As already mentioned, the migration protocol uses also a checkpoint as part of the connection status. This checkpoint is built at the connection setup time and stores incoming packets. It is cleaned up periodically as the request processing is carried out. Its main role stems from the fact that, most of the time, a migration decision comes as a consequence of analyzing a given request sent by the client along the connection. Since the outcome of that analysis is to migrate the connection endpoint to some other server in the cluster, the migration protocol has to deal also with the packet(s) of the request(s) and, possibly, with earlier packets (i.e., those that haven't been processed yet) or later packets (i.e., those arrived after the migration decision has been reached) as well. Thus, when migrating a connection, the content of the checkpoint is flushed out to the new location of the migrated endpoint. There, it is replayed in order to mimic the receive of the packets at that

node. In turn, the user-space daemon carries out the application-level protocol.

During the replay of the checkpoint at the remote site, the automatically generated ACKs of the TCP engine are dropped locally. The reason for this decision is performance- rather than correctness-related. Since the packets gathered in the checkpoint have been acknowledged once by the old server, the new server doesn't want to fool the client by sending duplicate ACKs as these may trigger on the client side the congestion avoidance [62] algorithm. As a result of that, the future data transfers will be erroneously penalized in terms of performance, since no congestion took place, actually.

6.6.5 Isolating the migration protocol from the client

During the migration process, the node initiating the migration suppresses all the packets that may be sent to the client (including the simple ACKs sent automatically by the TCP engine in response to the client packets received after initiating the migration). As soon as the two back-end nodes agreed on the migration and the duplicate connection endpoint has been fully set up at the new site, the new node is also responsible of sending to the client the packets from the write queue on behalf of the old server.

6.6.6 Updating the front-ends after a migration

When using the front-ends for routing , as soon as the migration completes, each ACK sent by the new server acts as an update sent to the front-end in order to modify its corresponding connection routing cache entry, if any. The front-end notices that it is an update message, and, if no entry for the connection exists, it registers the mapping between the connection and the MAC source address of the outgoing packet. As soon as the first client ACK arrives at the new server, the updating process stops (that is, server ACKs act no more as update messages).

Piggybacking update information on regular TCP ACK messages is possible because of the design of our server that requires the front-ends to be directly linked to the back-ends through a LAN. In this case, it is possible to set the type of the LAN packet carrying the ACK segment to be that of an update message. When the front-end receives and recognizes such a message, it will update first the connection-to-MAC address mapping table and then will route the packet further.

6.6.7 Setting up the forwarding table when no front-ends are used

When no front-ends are involved, as soon as the migration completes, the server initiating the migration registers locally the connection identity (client IP and TCP

port) and associates it with the server ID of the new location of the server-side endpoint. This routing entry is entered in the forwarding table whose role has been previously explained in Section 6.5. The forwarding takes place also over the SAN in order to minimize the latency and to avoid increasing the traffic on the LAN.

6.6.8 Handling the migration failure

If the new server does not accept the migrated connection endpoint, a fall-back mechanism is used to resume the execution at the old server. A *migration reject* packet is sent back and recognized by the old server as an error condition. Therefore, the server policy that triggered the migration is signaled the error. In turn, this policy will have to reach a decision. Normally, the request processing is resumed from where it was left. Anyway, as soon as the server has been informed about the abort of the connection endpoint migration, the connection is viewed again as a regular, locally-bound connection.

6.7 Operating system features that influence the protocol

There are several implementation aspects that influence the protocol design. First of all, TCP uses time stamps to prevent either parties to receive stale packets. These time stamps are most of the time taken from the local logical clock (i.e., an integer counter) of the operating system (in Linux, the so called *jiffies*). Currently, there is no way to synchronize these internal counters in a cluster (because they have internal relevance only). When it comes to migration, if the old server uses “newer” time stamps than those of the new server accepting the migrated endpoint (i.e., the old logical clock is greater than the new one), the end result will be that the TCP engine of the client will refuse the packets coming from the new server because they have “older” time stamps (i.e., smaller than those expected by the client that considers the corresponding TCP segments to be stale). In order to save extra synchronization messages, the value of the old server’s clock is piggybacked on the migration SYN as well by using one of the SYN option fields [41]. At the remote node, the value is recorded in the environment of the socket representing the migrated connection endpoint. A filter installed on the outgoing TCP path in the kernel checks every migrated connection endpoint environment for the value of the “old” time stamp and adjusts accordingly the time stamps in every outgoing TCP segment.

Further concerns address the TCP connection setup protocol. First, the TCP engine implementation in the kernel limits the number of outstanding connection requests that have not yet been accepted (i.e., the size of the SYN_Q queue, see Subsection 6.2.3). Since the connection endpoint migration emulates a client con-

nection setup protocol, it means that the migration requester competes with actual clients for the resources of the migration target node. Nevertheless, the migration implies more than the connection setup and requires a special treatment if the migration request is to be refused. Second, for performance reasons and in order to minimize the SAN traffic, the connection setup part of the migration protocol sends just the SYN segment over the SAN, while the corresponding SYN_ACK and ACK are played locally by the migration target. This solution makes the old server appear as a potential threat to the new server which may mistake it for a misbehaving client either attempting to conduct a SYN flooding attack or being too fast to respond to and therefore asking to be quenched. The solutions to these problems are discussed in Subsection 6.7.1.

Finally, in order to offload the LAN from additional traffic, the migration protocol takes place entirely on the SAN. This choice is natural because SANs have lower latencies and higher bandwidths than LANs. The protocol runs mostly in interrupt context (*bottom halfs* in Linux). Only the decision to initiate a connection endpoint migration cannot be taken in interrupt context because of two reasons. First (and obviously), the policies taking such decisions act on behalf of user processes (server programs) when these run in kernel mode. Second (and consequently), those processes must be stopped during the migration as the use of polling inside the kernel may harm the system performance (Linux is a non-preemptive kernel, at least the versions before 2.6). However, blocking the process involves a *sleep* mechanism that cannot be used in interrupt context inside the kernel.

6.7.1 The consequences of using the three-way handshake setup protocol for the connection endpoint migration

Using a modified connection setup protocol to migrate connection endpoints has the advantage of being easy to understand. Using the state of a given connection gathered in the connection checkpoint, the protocol simulates a connection establishment and then re-plays remotely the checkpoint. Moreover, it is easy to implement as it requires only to send the right SYN packet and to flush the checkpoint content to the remote site.

However, there are some catches as well. Depending on the processing speed of the requests and the rate of the request arrivals, it may very well happen that the SYN_Q of the new server overflows. When the overflow occurs, two decisions come in handy to the kernel. The simplest one discards the incoming SYNs. The more refined one requires SYN *cookies*. In both cases, the connection endpoint migration fails. A *reject* message is sent back to the requester to signal the failure. Therefore, our migration protocol may be sensitive to the size of the queue, recorded in the kernel as the SOMAXCONN value. This aspect is important as the

migration SYNs sent over the SAN backplane are competing for resources with the "regular" SYNs received over the LAN. As it will turn out, however, the migration rejection is not too costly and the extra percentage of rejected migrations results in insignificant penalties on the user-perceived latency and throughput. Nevertheless, the request distribution schemes relying on such implementations of the connection endpoint migration mechanism should be aware of this effect.

Using the connection setup protocol to duplicate the server-side endpoint on a remote machine has another drawback as well, besides the competition of the migration SYNs with the client SYNs for the resources of the server targeted by the migration. The migration SYNs use the SAN to fulfill the setup protocol. The SANs are high speed interconnects and, if many migration requests hit the new server, this one gets the false impression that eager "clients" are overwhelming it with requests. The phenomenon is unfortunately accelerated by our design decision to drop the SYN_ACK and to respond on the spot with the corresponding ACK. The TCP engine of the server simply drops these locally generated ACKs by advertising a zero receive window (see RFC 793 [41]) for the freshly allocated socket. As a result, the initiator of the migration doesn't get the migration acknowledgment and a deadlock occurs. Fortunately, breaking the deadlock is simply a matter of modifying the TCP engine to impose a non-zero receive window (64 KB by default) on sockets freshly created as a result of a connection endpoint migration.

6.8 The use of the TCP connection endpoint migration in request distribution policies

Having described the connection endpoint migration protocol, we now present the operation of the request distribution policies using it. We first discuss a simple policy aiming at assessing the overhead of the connection endpoint migration, both when it succeeds and when it fails. Next, we show how the connection endpoint migration can be used in a more complicated policy.

6.8.1 The simple policy

Essentially, this policy migrates all the requests hitting a certain node to another one in the cluster. From a technical perspective, this procedure works as follows. At interrupt time, the packets of a request build up in a receive queue of the socket. This queue is part of the checkpoint that the migratory connections manage. As soon as the request is reconstructed in memory, the server application can access the request data through *read/readv* or *recv/recvfrom/recvmsg* system calls. Inside the Linux kernel, a *tcp_recvmsg* routine checks the receive queue and transfers the

data to the user level. Our policy operates as a hook inside *tcp_recvmsg* by avoiding the data transfer to the local server in order to migrate the connection endpoint. Section 6.9 shows the evaluation results.

The signature of the hook inside *tcp_recvmsg* is the following:

```
int tcp_recvmsg_hook(struct sk_buff *skb);
```

A policy can be expressed by implementing this hook inside a Linux kernel module and becomes effective by downloading the module into the kernel by means of regular Linux *insmod/modprobe* commands. The parameter of the call is a socket buffer representing the packet about to be delivered to the user-level server program. By parsing the content of this packet (and possibly that of the subsequent ones), a policy can identify the requests of the protocol it was devised for (HTTP, for our experiments). The return value of the *tcp_recvmsg_hook* call specifies whether the packet should be handled locally (otherwise said, if the migration failed or hasn't been attempted at all) or the connection endpoint was migrated.

By the time *tcp_recvmsg_hook* is called, the corresponding socket buffer passed as an argument to the call is already associated with a socket representing the local connection endpoint. This information can be then used as a parameter to the migration routine, if the policy decides to migrate the connection endpoint. The signature of the migration routine is:

```
int migrate_sock(struct sock *sk, int target_ID);
```

The first parameter is the socket representing the local connection endpoint, while the second argument is the node ID of the server (provided by the communication infrastructure, see Section 2.4) to which the connection endpoint is being migrated. The return value specifies whether the migration succeeded or failed.

6.8.2 Handling short-lived connections

In a more general context, the policy might parse the request to find out information that might help take a request distribution decision. For instance, a content-aware policy might find out the file to be accessed and, depending on the actual location of the cached copies of the file, a decision meeting locality of reference demands may be taken. Such a solution applies of course to the cases where the application level protocol doesn't involve user-level specific processing such as compression or cryptography. For such cases, the policy must be implemented in user-level, while the connection endpoint migration is supported through a system call. However, an user-level policy implies changing the application software, which breaks our aim of running unmodified software designed for stand-alone infrastructures on top of clusters.

In this subsection we describe a particular type of policy that builds upon previous research. Crovella et al. [22] and Harchol-Balter et. al [34] proved that favoring the short-lived connections in connection scheduling for stand-alone servers improves significantly the average response time, practically without affecting the response time of the long connections. We wrote a policy that extends this result cluster-wide, by migrating only the connections concerning the small and popular static Web documents in a content-aware dispatching scheme. Since the large document requests are serviced locally, it can be considered that the short-lived connections are favored by migrating them to a site where they can benefit of improved locality of reference. At the node initiating the migration, we hide the costs of migrating the connection endpoints for the small requests by overlapping the increased disk I/O (due to the local service of the large files) with the computation.

As described in the previous subsection, we download a specific routine into the kernel (more exactly, within *tcp_recvmsg*) to do the job. This routine parses the HTTP protocol header, identifies the file name of the requested document and checks its size. If this size doesn't qualify the document to belong to the "heavy tail" of the request distribution curve, the policy migrates the connection endpoint to a node whose ID matches a hash value on the file name. This decision aims at creating locality of reference for a certain file by constantly caching its content on the same machine. Section 6.10 shows the evaluation results for this policy. Naturally, such a particular policy could have been implemented at the front-end level as well, but in a more complicated policy involving cooperative caching, for instance, the migration destination would be chosen according to caching hints taken from the cooperative caching management system. When used in such schemes, the front-end dispatching needs explicit caching hints from the back-ends.

6.9 S-Clients performance evaluation

To evaluate our simple policy using the connection endpoint migration (see Subsection 6.8.1), we ran a benchmark depicting the server performance in terms of connection throughput. We used S-Clients [10], a software that generates bursty HTTP request behaviors as typically seen in the Internet. The operation of S-Clients tests only the speed of the server software (including of course that of the kernel TCP/IP stack) without paying attention to additional issues like caching. The benchmark generates high rates of HTTP requests targeting a single file, *index.html* (by default, roughly 5 KB in our experiments, if not otherwise stated).

A client machine (called C) launches the benchmark. The generated requests pass through a front-end and are directed to a server machine called A. At this point, we explored three scenarios. In the first one, called **Single server**, the machine A serves the requests itself. This scenario serves as a baseline case for the

comparisons with the next two scenarios. In a second scenario, **Front-end**, the server A accepts the requests and migrates the corresponding connection endpoints to another server (called B) using the front-end based version of our migration protocol. B accepts the requests and serves them. The third scenario, **Back-end**, is similar to **Front-end** but we use the migration protocol that involves back-ends only. The general idea of the last two scenarios is depicted in Figure 6.3.

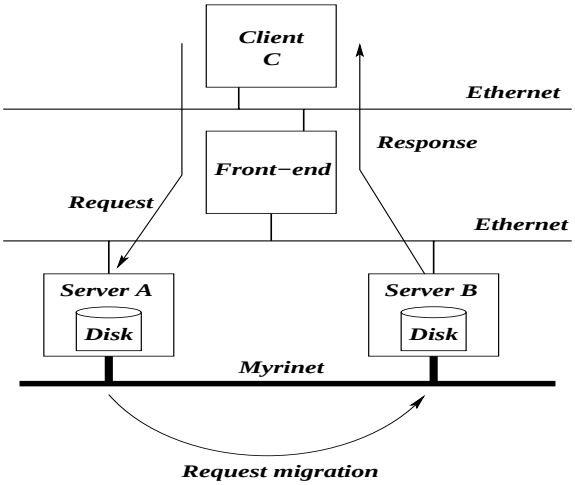


Figure 6.3: Experimental setup for connection migration policies migrating requests from one back-end server to another

We tried also to assess the impact of our particular protocol design on the connection endpoint migration performance. One such design aspect is the use of the three-way handshake protocol to initiate a migration. A migration can be refused by the remote server by usual means of rejecting a connection setup request (see Section 6.2.3). We assess the performance of the worst case scenario when all the migrations fail due to connection setup rejection at the remote server. In this case, the server B rejects all the migration requests of A, and, therefore, A has to resume the request servicing from the point it was left before the migration. This scenario intends to evaluate the operation of the connection endpoint migration when wrong migration decisions hit overloaded servers that cannot accept further requests.

6.9.1 Experimental setup

The two back-end servers, A and B, are the PCs described in Subsection 3.10.1. They are interconnected by means of the Myrinet infrastructure presented as well in Subsection 3.10.1. We used the GM 1.6.4 [66] version of the Myricom driver. Each back-end runs Apache 1.3.28 [5] as a Web server.

The client C and the front-end are both PCs equipped with Athlon AMD XP 1.5

Ghz processors and 512 MB of RAM. Both systems run Linux 2.4.19. All the machines, including the servers, are interconnected through regular 100Mb/s Ethernet (with the front-end acting as an IP router between the client and the servers).

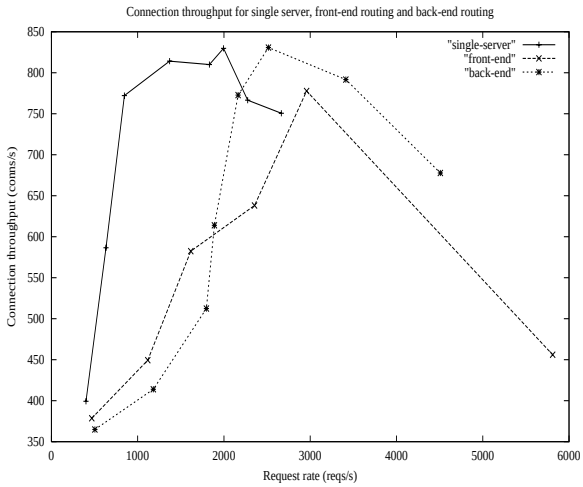


Figure 6.4: Server connection throughput

If the connection endpoint migration is to be used as an active element of a request distribution scheme, it needs to perform reasonably well under overload conditions. Therefore, our experiments aim to shed light on the performance of the connection endpoint migration under such circumstances. In order to understand the following results, we need to clarify first the notions S-Clients operates with.

6.9.2 S-Clients

The S-Clients benchmark enables the clients to generate variable request rates depending on some parameters. One of them is the number of the CPU cycles after which a request is issued by the client. This value is computed by dividing the CPU clock speed by a factor called from here on *rate*. The benchmark allows varying this *rate* and we used that in our experiments to increase the load on the server.

Another parameter that helps create high request rates is the number of repetitions of the attempts to get a request through (i.e., those requests which have been served by the server). Practically, this parameter translates into the lower bound on the number of successful transactions which have to occur before the benchmark stops. In our experiments we used 30000 repetitions.

By connection throughput we understand the number of connections per second accepted and served by the server. The request rate is the number of requests arriving at the server (i.e., the numbers of SYNs/sec).

The request rate generated by S-Clients is dynamically adapted to the server response. That is to say, for a given set of input parameters (CPU cycles, rate,

number of repetitions), S-Clients may generate different request rates for different servers or different setups of the same server. This behavior can be easily seen on Figure 6.4, where three different curves corresponding to different servers or to the same server with different setups are drawn by using x-axis values that differ as well. The explanation is that we used the x-axis to represent request rate values and, as mentioned before, the request rates are generated by the S-Clients depending on the server load and/or the network congestion. Alternatively, one can say that both the request rate and the connection throughput are *output* values of the S-Clients benchmark.

6.9.3 Connection throughput

The results concerning the three cases (**Single server**, **Front-end** and **Back-end**) are reported in Figure 6.4. **Single server** accommodates better lower request rates than the connection endpoint migration cases. For small request rates, the connection endpoint migration overhead is significant and, due to the low overall load of the server, the server performance is affected visibly. However, as soon as the request rates grow "enough" (beyond 2000 reqs/sec), the performance of the connection endpoint migration cases gets better than that of **Single server**. This result is due to the fact that **Single server** runs to its saturation while the connection endpoint migration cases are not there yet. So we shouldn't actually talk about improvement, but rather about the capacity of the connection endpoint migration to postpone the saturation point for higher request rates than those of **Single server**. For **Single server**, the saturation occurs at about 3000 reqs/s and the client running the benchmark exhausts the local resources and cannot create more connections as there are too many open connections to which the server didn't respond yet. Figure 6.5 shows a similar situation that will be discussed in the next subsection.

	Back-end	Front-end
# of migrated connections	209776	140336
# number of rejected migrations	100873	59955
Total # of connections	310649	200291
Successfully migrated ratio	67.52%	70.06 %

Table 6.1: The connection migration success rate

These results are good news for the request distribution, as the connection endpoint migration makes little sense for lightly loaded server nodes. Instead, it should be used to postpone the server saturation point. Notice that this goal corresponds exactly to that part of the curve where the connection endpoint migration pays off.

In Subsection 6.7 we discussed the influence of the kernel design on our choice of using fake SYN packets to initiate a migration. In Table 6.1 we quantify the

migration success of the two experiments using the connection endpoint migration. It can be seen that the back-end routing has a somewhat smaller migration success rate than the front-end routing (67.52% vs. 70.06%).

6.9.4 The evaluation of the migration rejection impact

A worst case scenario evaluation completes the image of the performance of the connection endpoint migration. Figure 6.5 depicts the performance of the server A, both stand-alone and while having all its migration requests rejected by the server B. Notice that the "rejection" curve on the graph depicts the performance of a single machine, namely that of the server initiating the migrations (the server A).

As it can be noticed, the rejection scenario is similar to the aforementioned connection endpoint migration scenarios in the sense that it doesn't hurt the performance of overloaded servers. That is to say, overloaded servers can try to migrate requests without being afraid that they will pay too much for their decision.

In order to get more insight on the causes of the overhead induced by the constant rejection of migrations, we draw an additional curve on the graph in Figure 6.5, namely the one labeled "delayed". Essentially, this curve describes the performance of a stand-alone server using a policy that chooses to delay by one CPU quantum the service of an incoming request. In terms of the policy implementation described in Section 6.8, we call the *sleep_on_timeout* kernel routine inside *tcp_recvmsg* with a timeout value of 1 CPU quantum (i.e., 1 *jiffy* in Linux).

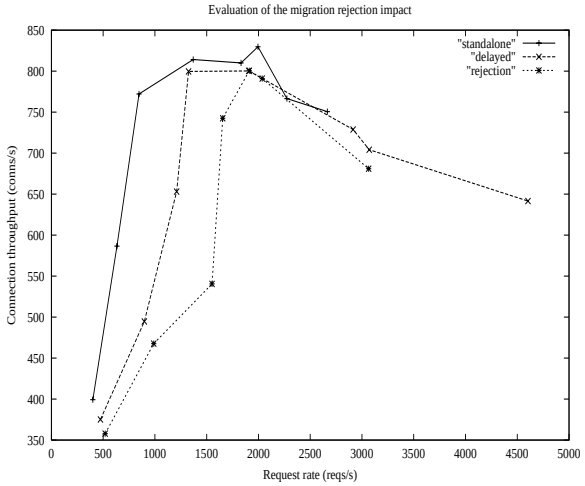


Figure 6.5: Evaluation of the impact of rejecting connection migrations

We said in Section 6.7 that the process initiating the migration has to yield the processor until the migration either completes or gets rejected. Therefore, the "delayed" curve aims to depict the behavior of a migration policy excluding the migration protocol itself. In fact, we are looking for an assessment of the network

costs of sending the migration SYN packet and having to handle the incoming migration rejection response. This assessment can be inferred by taking the difference between the "delayed" and the "rejection" curves.

The main observations are that, for low request rates, the network overhead of the migration is significant, while for high request rates, this overhead can be hidden through more request processing. Such results are typical for overlapping communication with computation, a technique that the kernels perform by default to improve the CPU usage by means of multiprocessing.

6.10 The WebStone performance evaluation

In order to assess the performance of the policy migrating short-lived connections (see Subsection 6.8.2), we used the WebStone [61] benchmark under the same assumptions described in Section 5.3 with one difference: the data set was replicated on the local disks of each of the two servers. We used the same experimental setup like that of the simple policy (see Subsection 6.9.1).

# simultaneous connections	100	150	200	250	300
Avg. response time RR (msec)	310.3	332.7	370.3	453.4	1113.9
Avg. response time CM (msec)	301.2	311.9	341.8	747.5	1615
Avg. throughput RR (Kbits/sec)	385.0	359.5	323.7	264.5	107.6
Avg. throughput CM (Kbits/sec)	397.2	384.0	351.1	159.9	74.2

Table 6.2: WebStone overall average response time and throughput figures for migrating non-persistent HTTP requests for small, popular Web documents (class0 and class1) vs. Round Robin routing

On such a setup, the policy described in Subsection 6.8.2 functions as follows. The client machine C runs the WebStone benchmark and generates requests that are redirected by the front-end to one of the back-end servers (the machine A). This server identifies the requests for the large and unpopular static Web documents (class2 and class3 as seen in Section 5.2) and services them locally on the server A, while routing the class0 and class1 requests to the second server (the machine B). In turn, the server B serves the requests. With the exception that only part of the requests are getting migrated, the operation is the same with that described in Figure 6.3. We compare the performance of this server setup with that of a two-node server handling requests routed in a Round Robin fashion by the front-end.

6.10.1 The evaluation of non-persistent HTTP connections

In a first round of experiments, our WebStone client issued plain HTTP 1.0 [13] requests without using the "Keep alive" feature of the protocol (see [30]) in order

to ensure that only one request is passed along any given TCP connection (i.e., we used non-persistent connections to the server). The reason to do so was that we wanted to be able to observe the caching behavior of our server using the aforementioned policy. The overall results are presented in terms of average response time and throughput in Table 6.2. A more refined analysis of the results for 100, 150 and 200 simultaneous connections is possible by having a look at Figures 6.6 and 6.7, which break down the overall figures into the corresponding class figures. For 250 and 300 simultaneous connections, the server evolves towards its saturation point both for Round Robin and our connection migration policy. However, our policy does worse than Round Robin because of the connection endpoint migration overhead. Both cases are discussed at the end of this subsection.

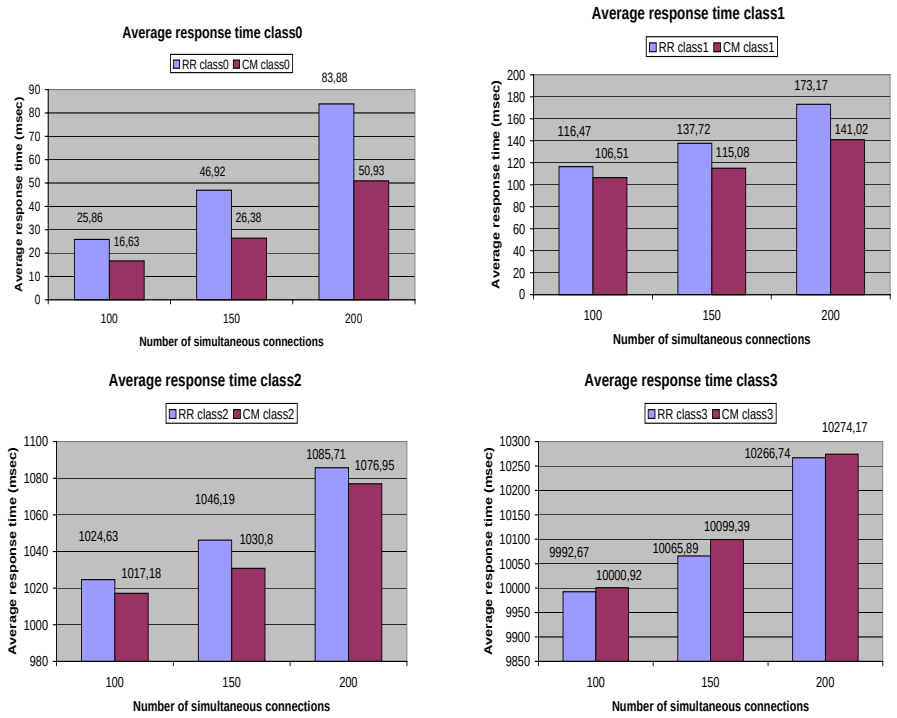


Figure 6.6: Average class response times for WebStone non-persistent HTTP requests

A look at Figures 6.6 and 6.7 shows that reassigning the service of class0 and class1 documents through connection endpoint migration improves the average response time and throughput for these individual classes. Indeed, one notices that, for class0, for instance, the average response time of the connection endpoint migration policy experiences reductions of 35.69%, 43.77% and 39.28%, respectively, of the average response time for the Round Robin policy. A somewhat less

severe degradation for Round Robin is exhibited by the average response times for class1 (8.55%, 16.43% and 18.56%, respectively).

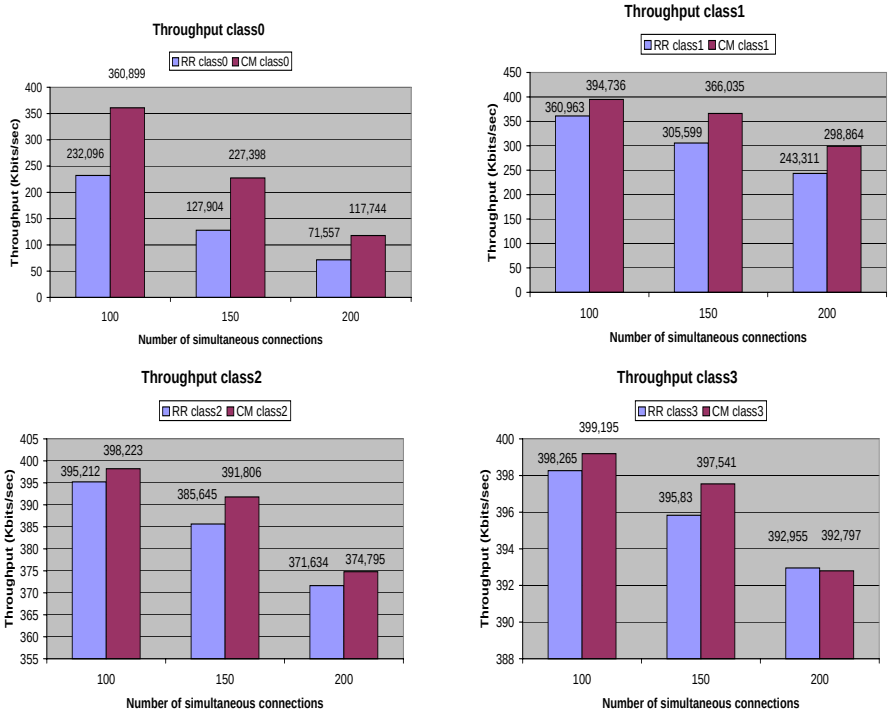


Figure 6.7: Average class throughput for WebStone non-persistent HTTP requests

All these benefits come practically at no extra servicing cost for the large and unpopular request classes (class2 and class3). In fact, the class2 figures show that the connection endpoint migration policy slightly outperforms Round Robin. For class3, one can notice an insignificant degradation of the average response time figures. However, the average response time degradation doesn't go beyond 0.03% of the corresponding connection endpoint migration figure.

These results confirm cluster-wide those obtained by Crovella et al. [22] when using a Shortest Connection First connection scheduling policy in stand-alone Web servers. In our case, the scheduling is done by migrating the requests for the small and popular files to the second server. The practically unaffected performance for the classes of large and unpopular documents has several reasons. First, as explained in the previous chapter (see Subsection 5.1.1), the Zipf-like distribution is not an exponential one and thus makes the requests for the large documents seldom enough. As a result of that, such requests do not to have an impact on the servicing time when migrating the requests for the other classes. Second, the long servicing time for large and unpopular documents makes it easier to hide the connection

endpoint migration overhead of the requests for small files through an increased overlapping between computation and I/O (disk and network altogether). And naturally, the high popularity of the small documents as well as their reduced size make the caching at the migration site very effective for these classes of requests.

Overall, the individual improvements for the classes of small and popular requests are somewhat smoothened out, as reflected in the average figures shown in Table 6.2. Indeed, the overall improvements in terms of the average response time, for instance, amount, to 2.93%, 6.25% and 7.69%, respectively.

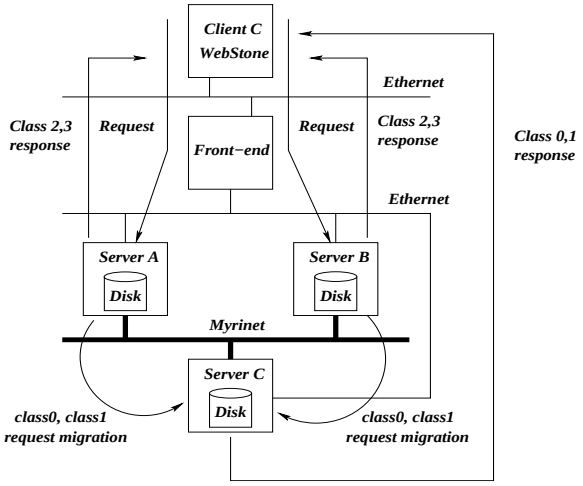


Figure 6.8: Connection migration policy with three servers, two of which receive requests on a Round Robin basis and decide in turn to migrate those addressed to class0 and class1 documents to a third back-end server

# simultaneous connections	250	300	350
Avg. response time RR (msec)	416.1	907.0	1150.6
Avg. response time CM (msec)	410.3	513.2	820.4
Avg. throughput RR (Kbits/sec)	288.1	132.2	104.0
Avg. throughput CM (Kbits/sec)	291.9	233.3	145.9

Table 6.3: WebStone overall average response time and throughput figures for three servers when migrating non-persistent HTTP requests for small, popular Web documents (class0 and class1) vs. Round Robin routing

As soon as the WebStone load increases beyond 250 simultaneous connections, the performance of our connection endpoint migration policy worsens, as it can be inferred from the last two columns of Table 6.2 which depict the overall performance of the two policies for 250 and 300 simultaneous connections, respectively. In order to pinpoint the problem, we decided to investigate whether the server A

(the one initiating the migrations) doesn't become a bottleneck of the system as the load increases. Since our suspicion was that the cause of the performance loss was the overhead imposed by the connection endpoint migration, we decided to scale up the server setup to three back-end servers and to distribute the connection endpoint migration task to two of the back-end machines.

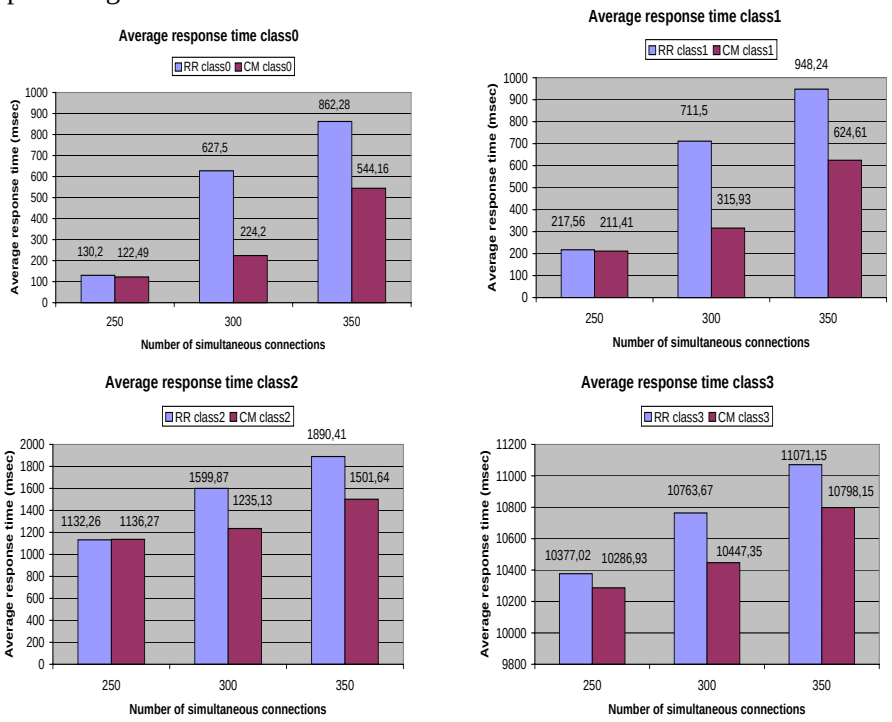


Figure 6.9: Average class response times for WebStone non-persistent HTTP requests in a three-node cluster-based server

6.10.2 The evaluation of non-persistent HTTP connections for a three-node cluster-based server

In order to answer the question, we use a slightly different server setup in which the front-end routes requests according to a Round Robin policy to two of the back-end servers, which, in turn, migrate class0 and class1 requests to a third back-end server. A visual description of the setup is provided by Figure 6.8. By distributing the migration task between two back-end servers, we attempt to balance the migration overhead between the two servers and to see whether this solution improves the cluster-based server performance. All the experimental results concerning this policy are then compared to those of a policy dispatching the requests to three servers on a Round Robin basis. The overall comparison of the two policies can be seen in Table 6.3, while Figures 6.9 and 6.10 show the per-class performance.

Figures 6.9 and 6.10 show indeed that the problem of the previous connection endpoint migration policy was that, for heavy workloads, the server migrating the connection endpoints couldn't cope with the connection endpoint migration overhead. By distributing this overhead almost equally over two servers, one gets a policy that performs significantly better than Round Robin as the load increases.

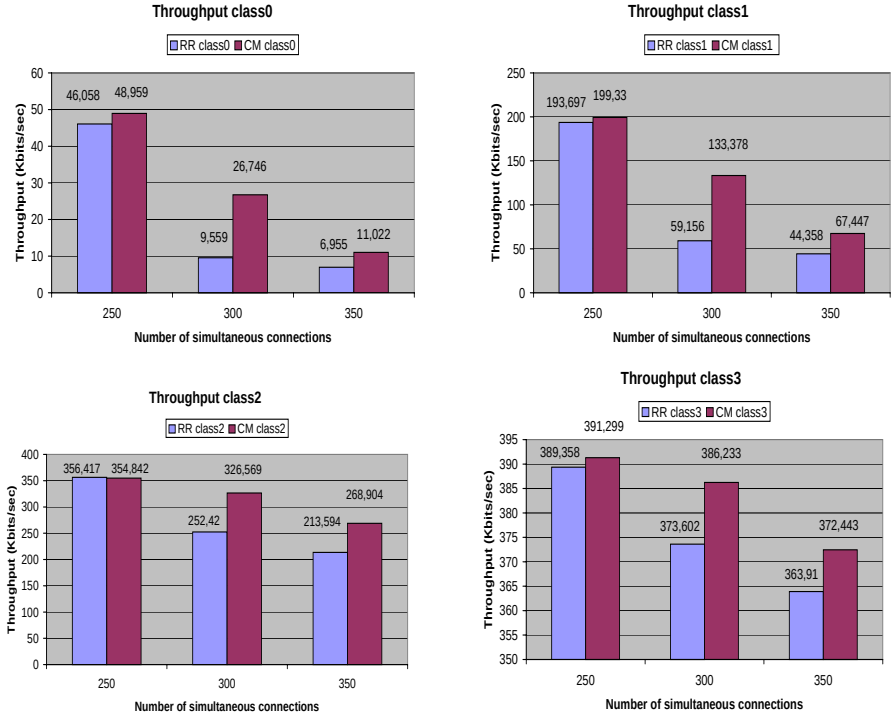


Figure 6.10: Average class throughput for WebStone non-persistent HTTP requests in a three-node cluster-based server

The differences are not significant for 250 simultaneous connections; for this load, the connection endpoint migration policy does slightly better than Round Robin. However, as soon as the load increases to 300 and 350 simultaneous connections, the connection endpoint migration policy outperforms clearly Round Robin and copes better with the performance degradation. A look at Figure 6.9 shows that, for 300 connections, the class0 average response time of the connection endpoint migration policy is 2.79 times smaller than the corresponding Round Robin time, while for class1 the ratio is "only" 2.25. Even the class2 requests benefit significantly as the average response time for the connection endpoint migration policy is 22.79% smaller than that of Round Robin (see Figure 6.9). For class3, the gain is insignificant, 2.93% (notice that the scale of the class3 graph is truncated at the bottom). Similar conclusions hold for the throughput figures (see Figure 6.10).

A look at Table 6.3 gives us the overall improvement, both in terms of average

response time and throughput. The average response time of the connection endpoint migration policy is 1.76 times smaller than that of Round Robin, while its throughput is 1.76 times larger.

Similar conclusions hold for the 350 simultaneous connections case as well, although both servers degrade rapidly towards their saturation points. Still, in terms of the overall performance, the average response time of the server operating under the connection endpoint migration policy outperforms by a factor of 1.40 the Round Robin driven server.

6.10.3 The evaluation of persistent HTTP connections

All the experiments described in the previous two subsections considered non-persistent HTTP connections (i.e., one HTTP request per established connection) to the cluster-based server. Under these circumstances, it is easy to reason about the caching effects of the two previously presented request distribution policies involving connection endpoint migration. However, a fair question asks to assess also the TCP connection endpoint migration's performance for persistent connections.

Unfortunately, the answer is hardly foreseeable for simple policies like those used before. The main reason lies in the fact that, as soon as the client passes many HTTP requests along a given TCP connection to the server, it is almost impossible to guarantee any caching effectiveness for the requested documents. Indeed, migrating requests addressed to a given class of documents to a given server node doesn't ensure a successful caching, as the client will continue to send along the same connection requests that potentially (and most probably) target other classes of documents as well. As a result, the cache of the node where the connection endpoint migrates becomes polluted and less effective.

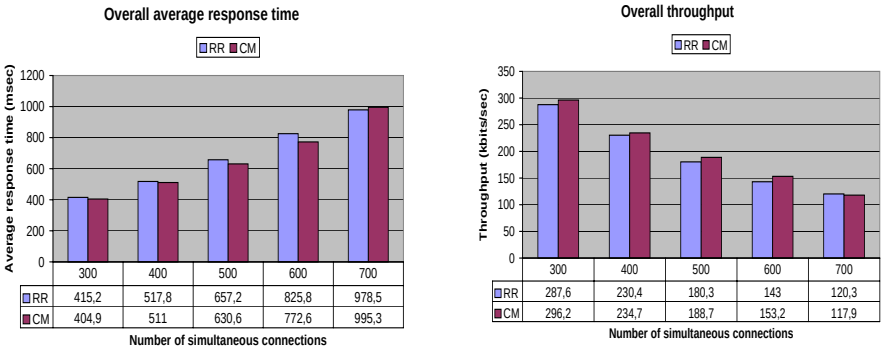
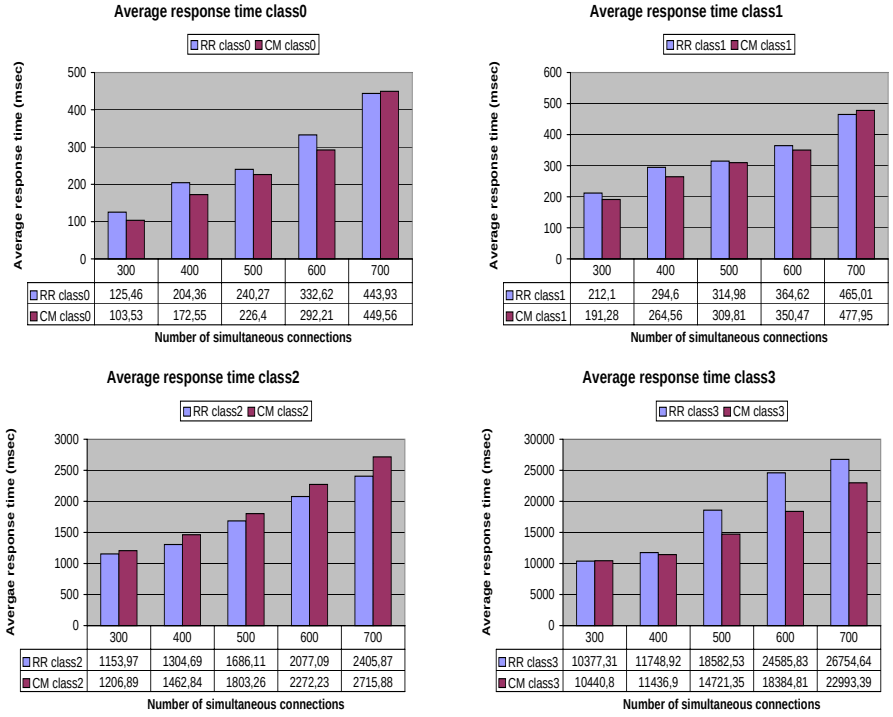


Figure 6.11: Overall average response time and throughput figures for WebStone persistent HTTP requests when migrating class2 requests

A simple solution of the problem would be to migrate the connection endpoint every time a new request comes along the persistent connection to a server node

that already caches the requested document(s). However, this solution barely fits our small experimental setup. With only two (or three) nodes, using such a policy results in thrashing, as the nodes spend most of their running time handing each other connection endpoints. Therefore, we stuck with the simple policies that we used in the previous two subsections and we detected empirically which classes of documents benefit at most from caching, as a result of the connection endpoint migration in the case of persistent connections. We consider this approach sufficient, as our goal in this subsection is only to prove that the connection endpoint migration performs well for persistent HTTP connections too.



	300	400	500	600	700
RR class0	125,46	204,36	240,27	332,62	443,93
CM class0	103,53	172,55	226,4	292,21	449,56

	300	400	500	600	700
RR class1	212,1	294,6	314,98	364,62	465,01
CM class1	191,28	264,56	309,81	350,47	477,95

	300	400	500	600	700
RR class2	1153,97	1304,69	1686,11	2077,09	2405,87
CM class2	1206,89	1462,84	1803,26	2272,23	2715,88

	300	400	500	600	700
RR class3	10377,31	11748,92	18582,53	24585,83	26754,64
CM class3	10440,8	11436,9	14721,35	18384,81	22993,39

Figure 6.12: Average class response times for WebStone persistent HTTP requests

In the rest of this subsection we present the results of using the policy described in Figure 6.3 when migrating requests for the static Web documents of class2 (large and unpopular documents, 14% of the total WebStone requests). As a short reminder, that policy operates on a two-node cluster-based server by migrating connection endpoints from one server node to the other one. As previously explained, choosing to migrate requests for class2 is simply based on the empirical results that are presented in Figure 6.11. Figures 6.12 and 6.13 present again class-based breakdowns that help understand easier the caching behavior of the server nodes for particular classes of the requested documents, both in terms of the

average response time and the achieved throughput.

First of all, one important aspect should be noticed by looking at any of these graphs. The use of persistent HTTP connections improves significantly the operation of the server (regardless whether it uses Round Robin or connection endpoint migration policies). Whereas in the case of the non-persistent HTTP connections our server saturates at about 300 - 350 simultaneous connections, the use of the persistent HTTP connections allows the same server postponing its saturation point beyond 700 simultaneous connections (up to 1000 as we will see next, in the case of the three-node server setup). The performance difference in the case of the persistent HTTP connections can be accounted to amortizing the cost of the connection setup and termination over multiple requests.

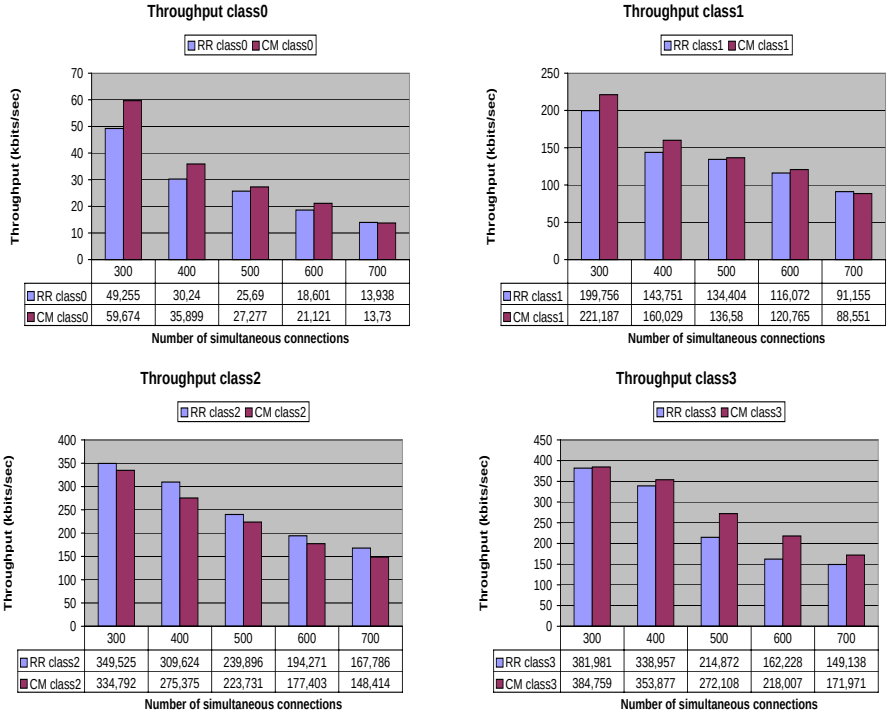


Figure 6.13: Average class throughput for WebStone persistent HTTP requests

By looking at Figure 6.11, one can infer that migrating requests for class2 balances the two-node cluster as well as a Round Robin policy does. Indeed, the overall performance figures of the connection endpoint migration policy are slightly better than those of Round Robin, except for the heavy load case (700 simultaneous connections) when the performance is slightly worse. An in-depth analysis of this result is possible by having a look at Figures 6.12 and 6.13. For instance, the graphs corresponding to class2 show the slight performance degradation due

to the connection endpoint migration overhead. The highest degradation concerns the 700 simultaneous connections case. In terms of the average response time, for instance, the penalty is an extra 12.88% of the Round Robin average response time for class2. In fact, for 700 simultaneous connections, Round Robin outperforms the connection endpoint migration policy for class0 and class1 documents as well.

However, in general, the performance figures for the class0, class1 and class3 requests are better for the connection endpoint migration policy. For class0 and class1, the gain is significant for the light loads (300 and 400 simultaneous connections). Again, reasoning in terms of the average response time, the best gain for class0 is 17.47% of the Round Robin time (for 300 simultaneous connections), while the similar figure for class1 is 10.19% (for 400 simultaneous connections).

The class3 figures of the connection endpoint migration policy are particularly good, as for all the loads they are better than those of Round Robin. And the interesting fact is that they are better as the load on the server increases. For the 600 simultaneous connections load, for instance, the servicing time of the class3 documents for the connection endpoint migration policy is with over 6 seconds faster (6201.02 milliseconds, in fact) than that of Round Robin, representing a save of 25.22% of the Round Robin time. Even for 700 simultaneous connections, the connection endpoint migration policy manages to save 3761.25 milliseconds, that is, 14.05% of the Round Robin time.

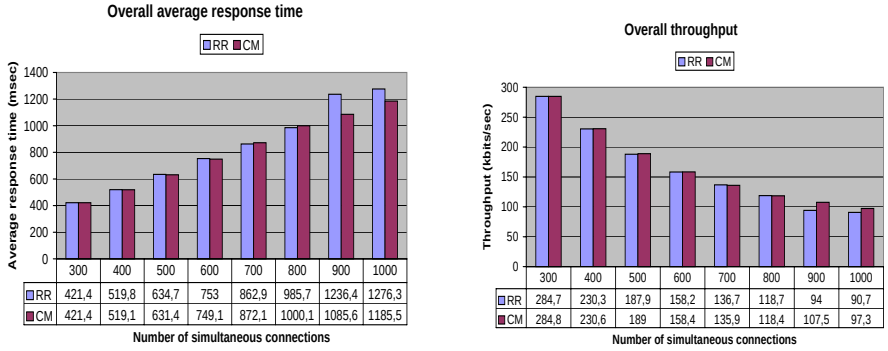


Figure 6.14: Overall average response time and throughput figures for WebStone persistent HTTP requests in a three-node cluster-based server

6.10.4 The evaluation of persistent HTTP connections for a three-node cluster-based server

In this subsection we present the results of using the connection endpoint migration for persistent HTTP requests in our second policy depicted in Figure 6.8. As a reminder, that policy operates on a three-node cluster-based server in the following way. The front-end router redirects in a Round Robin manner persistent HTTP

requests to two of the server nodes which, in turn, decide to migrate some of their connection endpoints to the third server. Unlike Subsection 6.10.2, the policy does not migrate class0 and class1 but class3 requests. As in the previous subsection, class3 has been identified experimentally to be the best-performing case.

The overall results are shown in Figure 6.14, while the detailed, per-class performance is depicted in Figures 6.15 and 6.16, respectively. The graphs in Figure 6.14 tell that the connection endpoint migration policy migrating persistent HTTP requests addressed to class3 performs as well as Round Robin does. For the heavy loads, 900 and 1000 simultaneous connections, respectively, it even outperforms the policy aiming at a perfect balance.

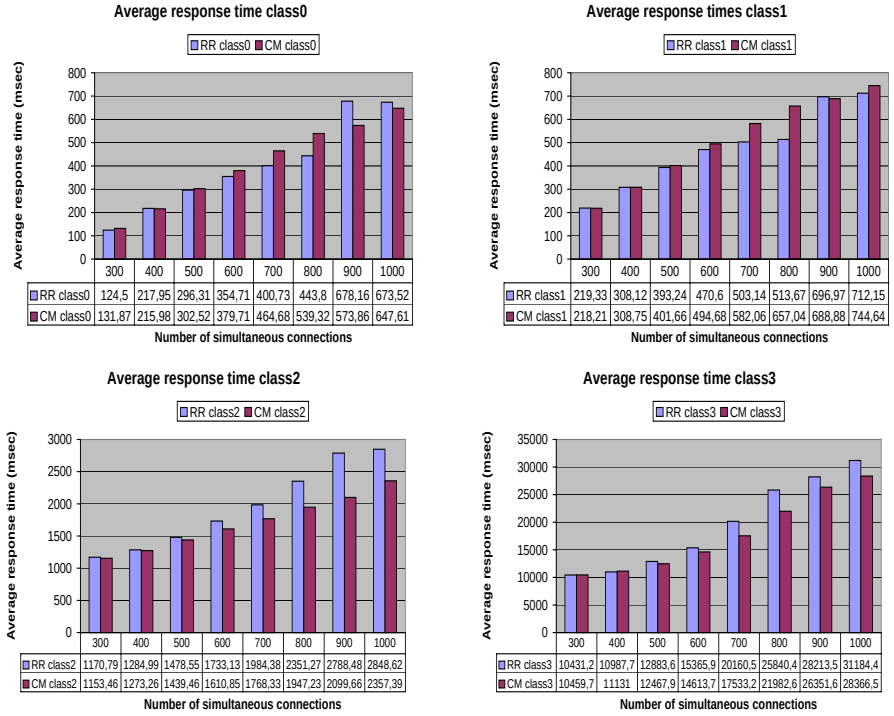


Figure 6.15: Average class response times for WebStone persistent HTTP requests in a three-node cluster-based server

A look at the graphs depicting the performance of the individual classes of requests explains this result. For all the loads except for the 900 and 1000 simultaneous connections cases, the connection migration policy outperforms the Round Robin one for class2 and class3, while exhibiting poorer performance for class0 and class1. For the last two loads, the situation changes. All the class results are better for the connection endpoint migration policy (except for the case of the class1 and 1000 connections, but even in that case the difference is minimal).

Moreover, the figures for class2 and class3, the large and unpopular documents whose servicing time accounts for most of the total servicing time, show clear improvements over those of the Round Robin policy. For instance, the connection endpoint migration policy operating on 900 simultaneous connections improves the average response time of the class2 requests by 688.82 milliseconds (24.70% of the corresponding Round Robin time) and that of class3 by 1861.99 milliseconds (6.59% of the corresponding Round Robin time). For 1000 simultaneous connections, the class2 average response time is with 491.23 milliseconds (17.24% of the corresponding Round Robin time) smaller than that of Round Robin, while for class3 the difference is of 2817.89 milliseconds (9.06% of the corresponding Round Robin time).

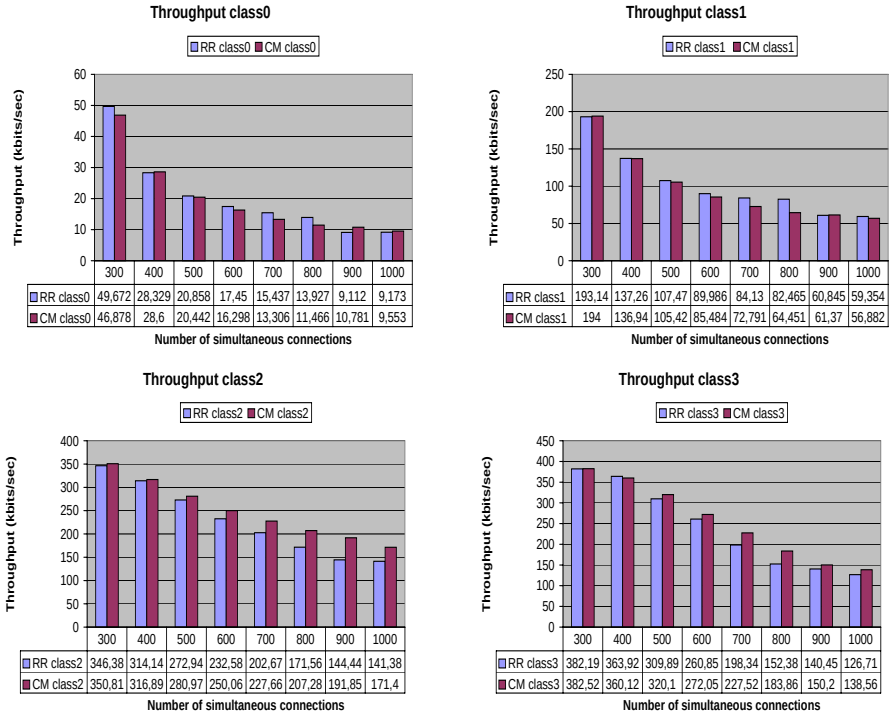


Figure 6.16: Average class throughput for WebStone persistent HTTP requests in a three-node cluster-based server

As in the case of the results presented in the previous subsection, the actual performance of the simple policies like those migrating entire classes of requests in order to improve the locality of reference for the requested documents of those classes is of little relevance for the persistent HTTP connections. Since many requests targeting various document classes flow along the same connection, assigning statically the migration destination based on the document class is hardly pay-

ing off in terms of the caching effectiveness. However, this section and the previous one point out the effectiveness of using the connection endpoint migration in back-end level request distribution schemes for persistent HTTP connections, provided that optimal caching offsets the incurred connection endpoint migration overhead. For our simple policies we had to find out empirically the optimal case, but it is perfectly possible to develop more sophisticated policies that strive to achieve an optimal performance by taking into account the caching information available at the back-end level. This type of information can be either explicitly disseminated throughout the cluster by the back-end nodes or implicitly shared among the back-end nodes by means of cluster-wide caching systems like that presented in the previous chapters.

6.11 Summary

We presented in this chapter a mechanism for migrating TCP connection endpoints between two server nodes in a cluster-based server. The main goal of our work was to assess whether the TCP connection endpoint migration can be an useful mechanism in the request distribution for cluster-based servers, as the previous research on client-server connection migration protocols showed that they can perform well as a fine-grain fail-over mechanism for fault-tolerant server solutions or as a support for host mobility and migration of server sessions.

The results of our experiments are encouraging for our endeavors. The connection endpoint migration shows throughput figures that adapt well especially for high request rates, which is important as the connection endpoint migration is envisioned to play a load balancing role by offloading highly loaded nodes from the request service. Even the extreme case of the constant migration rejection doesn't add a significant overhead to a system experiencing high request rates, which means that the connection endpoint migration can be trusted for highly loaded nodes as the penalty on the initiator won't be excessive.

We show also that the connection endpoint migration helps build simple but effective locality-aware request distribution policies. For instance, a simple policy that migrates all the non-persistent HTTP requests for small and popular static Web documents from one back-end server to another performs better than a Round Robin routing policy involving the two servers, both overall and for the considered classes of requests. The average response time for small requests gets improved by as much as 43.77%, practically at no extra costs for the "heavy-tail" service.

Moreover, the experiments with a three-node server distributing the migration task between two servers in a Round Robin manner, while, in turn, the two servers migrate requests for the small and popular static Web documents to a third machine, show that the connection endpoint migration policies can outperform simple

policies like Round Robin by a factor as high as 1.76. The individual figures for the migrated classes of requests (class0 and class1 in WebStone terminology) show improvement factors of 2.79 and 2.25, respectively, while improving the individual service figures of the other classes of documents at the same time!

The aforementioned results are possible because the non-persistent HTTP connections enable our simple policies to control the caching effectiveness of the server nodes. However, we show that the TCP connection endpoint migration can equally be an effective routing mechanism for persistent HTTP connections in request distribution policies operating at the back-end level. Nevertheless, in the case of the persistent HTTP connections, the performance of such policies is highly sensitive to the caching effectiveness of the requested documents.

The TCP connection endpoint migration involves only two cluster nodes at a time and therefore doesn't raise scalability questions. However, the use of the TCP connection endpoint migration in request distribution policies may raise such concerns. In fact, our connection endpoint migration policy operating on a two-node server (see Subsection 6.10.1) doesn't cope well with high loads, and only by distributing the overhead of the migration between two server nodes (see the policy described in Subsection 6.10.2) it was possible to obtain high performance results. So even for small-sized clusters, the inappropriate use of the TCP connection endpoint migration may have a negative impact on the performance. In general, the impact of the TCP connection endpoint migration on the scalability of the request distribution policies using it turns out to be an interesting and sensitive topic of further research.

Chapter 7

Speculative TCP connection admission in cluster-based Web servers

As seen in the previous two chapters, our SSI server relies on request distribution solutions taken at the back-end level. The main reason behind this design decision stems from the fact that the back-ends cooperate in order to serve requests (through cooperative caching, for instance) and thus share naturally various types of information about the other nodes in the cluster. Such an example is the way HSCC shares the load information among the cluster nodes in order to improve the eviction handling. Our policy-oriented, back-end level request distribution solution may thus profit easily from this kind of information that is locally available on each back-end node.

Having taken such a design decision, one has to look for appropriate mechanisms to support the back-end level request distribution. So far, we have seen in Chapters 5 and 6 how cooperative caching and the TCP connection endpoint migration function in our SSI cluster-based Web server. More complicated policies than the one presented in Subsection 6.8.2, for instance, would try to combine the cooperative caching with the connection endpoint migration at the back-end level and, by doing that, would need a particular back-end level mechanism for the load balancing. The main problem when developing such a mechanism consists in the ability to decouple the server functionality of the back-end cluster node from that of the load balancing. More precisely, doing load balancing at this late stage has to cope with the fact that the incoming requests have reached their final destination and expect to find here service.

In this chapter, we investigate the speculative TCP connection admission, a

back-end level load balancing mechanism either targeting fully distributed cluster-based servers (i.e., without front-end) or acting as a companion to the front-end request dispatching. The speculative admission builds upon the TCP connection endpoint migration by speculatively accepting incoming requests on overloaded nodes only to migrate them further to less-loaded servers in the cluster. Sub-optimal routing decisions taken outside the cluster (either at the front-end(s) or by external entities at earlier stages in multi-tier server architectures) may direct a request to an overloaded server and that will cause the rejection of the request. Of course, if all the other nodes in the cluster are also overloaded, there is little room to improve the situation. However, as soon as there are less-loaded nodes, the speculative admission offloads the server by migrating the connections to these nodes and thus offsets the imbalance. From this perspective, the speculative admission acts as a load balancing mechanism or, for the servers using front-ends, leverages the front-end load balancing. Acting at the back-end level, the speculative admission has the advantage of being fully distributed. The speculative admission is a mechanism and not a policy. Identifying lightly-loaded nodes in the cluster and developing methods to disseminate this information throughout the cluster are no topics of this chapter.

Parts of the material presented in this chapter were published in [51].

7.1 Speculative connection admission

In the cluster-based servers that use front-ends, the requests are routed to the back-ends either uninformed or based on some application-level protocol analysis. In the uninformed case, as we have seen before, the request is routed to a server according to a hash function aiming to equalize the loads of the back-ends. By inspecting the content of the request, the informed routing provides a better amortized performance by taking into account additional information such as locality of the requested data. In both cases, however, the sub-optimal routing decisions may result in severe load imbalances among the back-ends. One legitimate question is whether we can do better once such a decision caused a newly incoming request to be sent to an overloaded server.

The fully-distributed cluster servers do not use front-ends to assign connections to the back-end machines. They usually rely on an external entity to perform that assignment (Round Robin DNS [16], for instance) or perform the dispatching themselves. In the first case, the external entities performing the routing are not aware of the distributed server status and thus may inflict severe load imbalances. One needs a mechanism at the back-end level to offset such imbalances. The servers performing the dispatching themselves are most of the time “virtual servers”, that is, distributed servers using a common IP address which rely on

MAC-level routing. However, this aspect makes them hardly amenable to content-aware routing. One needs a back-end level mechanism to correct the routing decisions that are sub-optimal from the data locality point of view.

The speculative connection admission tries to answer these issues. The idea is to accept additional incoming connection requests even if the server is highly loaded, provided that there are less-loaded nodes in the cluster willing to take over the servicing of these requests. Once the connection is established, the overloaded server hands over the endpoint it manages to a lighter-loaded node. This task is accomplished through TCP connection endpoint migration.

From a technical perspective, this idea is accomplished in the following way. When using the standard socket library, a server declares its interest in servicing requests in two steps (see also Section 6.2). First, it invokes a *listen* system call that establishes a queue in the kernel for the incoming connection requests. Then, through an *accept* system call, the server picks up established connections from that queue and passes on the corresponding connection handles to that part of the server program servicing the requests. The speculative connection admission enlarges the queue storing the incoming connection requests by a given increment. The "extra-accepted" connections are marked migratory. A request distribution policy will migrate these connections according to either cluster-wide load hints or the locality of the referenced data.

In general, the TCP connection endpoint migration can be regarded as the driving engine of the distributed request dispatching at the back-end level. Postponing the routing decisions until the requests hit the back-ends is an important issue for the solutions that strive to enhance the cooperation among the back-ends. Such approaches are suitable when the back-ends share cluster-wide information susceptible of improving the request distribution decisions: load information (expressed by means of the CPU utilization of the cluster nodes, the number of the established connections, the amount of the I/O activity or combinations of all these) for the load balancing policies or globally cached data for the content-aware routing. In this regard, the speculative connection admission extends further the capabilities offered by the connection endpoint migration and acts as a support for the back-end level policies that leverage the routing decisions performed outside the cluster. In the next subsections, we describe the implementation of the speculative connection admission in the kernel and we show how to use it together with policies in our request distribution system.

7.1.1 Operating system internals

To understand the speculative connection admission, we need to remember how the requests are handled from the kernel perspective. In Subsection 6.2.3, we saw

that, when a client sends a request to a server, its TCP engine sends a SYN packet to the remote peer. The server receives and processes the packet at interrupt time using the event-driven model described in Subsection 6.2.2. Namely, it responds on the spot with the appropriate SYN_ACK and adds an open-request structure to the SYN_Q queue associated with the listening socket. As soon as the client sends back the ACK to the SYN_ACK, the corresponding open-request structure at the server is marked "ready" and a freshly created socket is associated with it. Later on, when the user-invoked *accept* runs in kernel context over the listen queue, the newly established socket is passed on to the server application and the open-request structure is released.

Although a single queue (a linked list of open-request structures), this list is logically managed as if there would be two queues: a SYN_RCVD queue (storing requests for unacknowledged received SYNs) and an "accept" queue (storing requests associated with already established connections). The *accept* system call considers actually only the "accept" part of the queue.

7.1.2 Speculative connection admission policies

The speculative connection admission enlarges the "accept" part of the listening queue by a given increment. Thus, it establishes a percentage of the incoming accepted connections that will have to migrate. In a logical sense, we can speak of an additional "admission" queue. Practically, this queue is part of the enlarged "accept" queue. The extra-accepted connections will be migrated by a speculative connection admission kernel policy which operates as follows.

When the client ACK in the three-way hand-shake connection setup protocol arrives at the server, the TCP handler creates a new socket and tags it "connected". At this point, the speculative admission policy marks the socket migratory. The number of migratory connections varies dynamically and depends on the ratio between the accept queue increment (the size of the "admission" queue) and the size of the whole accept queue.

As with the policies described in the previous chapter (see Section 6.8), the speculative admission policy operates within the *tcp_recvmsg* routine of the kernel. If the corresponding socket was previously marked migratory, the local request handling is avoided and a connection endpoint migration takes place. One legitimate question asks though why the speculative policy should wait for the server application to try to process the request before migrating the connection endpoint. The answer is that meaningful decisions might be possible only by inspecting the request. For instance, a content-aware policy (like that in Subsection 6.8.2) might find out the file to be accessed and, depending on the actual location of the cached copies of the requested document, a performant decision may be taken.

However, all this comes at a price. Modern server programs use multiple threads (sometimes even processes, like in Linux's case) to serve the requests. Since the speculative policy operates only when such a server thread attempts to process a request from the socket receive queue, it means that, although the request will not be handled locally, a new servicing thread is spawned and scheduled for execution. Even though many server programs use pre-spawned threads in order to avoid the overhead of the thread creation, there remains the price of an additional context switch. For an exact image of how much this cost may be, the reader is referred again to the "delayed" curve in Figure 6.5, Section 6.9. The right answer to this problem is to use the speculative connection admission only for high request rates, when the impact of the additional context switch isn't that expensive. Luckily, that operation regime is exactly the one addressed by the speculative connection admission which is supposed to work on overloaded nodes in order to balance the loads of the cluster nodes. Using the speculative connection admission for lightly-loaded nodes makes little sense.

One other important issue regards the migration rate. Assuming that servicing a request takes more time than speculatively admitting and migrating it, it means that the node speculatively admitting connections may soon overflow the migration target. This issue is regulated by considering the admission queue as a window that shrinks and grows depending on the capacity of the migration target to accept migrated requests. As soon as the migration target rejects a connection migration, the node migrating speculatively accepted connections closes its window (i.e., makes the admission queue length equal to zero) and waits for a notification from the migration target before it starts again to accept connections speculatively. The node receiving the migrated requests sends back notifications to the migration initiator as soon as it finished servicing the requests. If these notifications get lost and the node using the speculative admission has a null window, then the node will grow its window back to the size of the admission queue after a certain number of local requests have been serviced (typically, this value is also set to the size of the admission queue, as this size approximates the number of the migrated connections that the migration target might have processed in the meantime. Naturally, this assumption holds for homogeneous clusters only).

7.2 Performance evaluation

We used again S-Clients [10] to evaluate the performance of the speculative connection admission. We reported the performance of our system in terms of the number of serviced connections per second (the connection throughput) relative to the achieved number of requests per second (the request rate). All the experiments used a front-end based server. The case of the fully distributed server can be sim-

ply derived from this one by considering the front-end to be the external entity that dispatches the requests.

Our experiments consist in launching two S-Clients processes on a client machine called C. The requests generated by the benchmark pass through a front-end and are directed to a server machine called A. At this point, we explored two scenarios. In the first one, the machine A serves the requests itself. This case serves two purposes: as a baseline case for the comparisons with the next scenario and also as a way to understand the impact of varying the accept queue size on the server software. In a second scenario, the server A uses the speculative connection admission to offload some of its connections onto the other server (B). This second scenario intends to evaluate the operation of the speculative admission in real-life conditions, as the server A handles a given number of connections and migrates the speculatively accepted ones to the server B. In order to assess the impact of the load imbalances that may occur in a cluster, we instructed the front-end to route requests asymmetrically. More specifically, in one experiment the front-end redirects two thirds of the requests to the server A and one third to the server B. We tested also the case when three quarters of the requests are routed to the server A and one fourth to the server B.

We used the same experimental setup like that described in Subsection 6.9.1. Each back-end runs Apache 1.3.20 [5] as Web server.

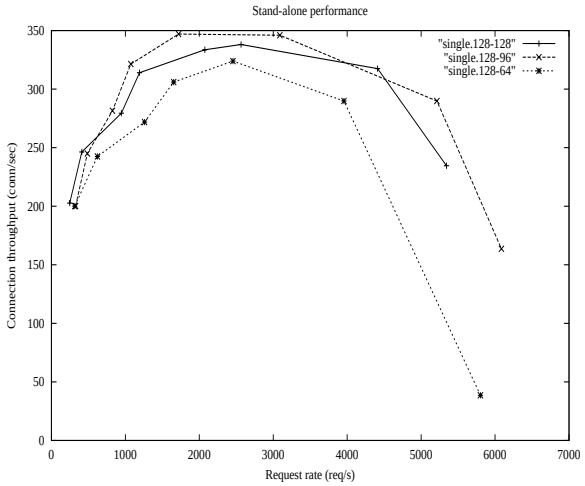


Figure 7.1: The impact of the accept queue length on the server activity

7.2.1 The impact of the accept queue length on the server activity

We started our experimental evaluation by testing the performance of a stand-alone server in terms of the achieved connection throughput when varying the size of

the "accept" part of the listen queue. The results are presented in Figure 7.1. The length of the SYN_RCVD queue was that set in the kernel as the SOMAXCONN value (128 by default). We chose length values of 64, 96 and 128 for the accept queue of our server (denoted on the graph by "single.128-64", "single.128-96" and "single.128-128", respectively). By looking at Figure 7.1, it can be noticed that the best performing case is "single.128-96", while the system experiences the largest performance degradation for an accept queue length of 64. Since we want to test the performance of the speculative connection admission under heavy load conditions, we chose the case "single.128-64" as a base case for our next experiments. That is to say, the speculative admission experiments considered a base accept queue of length 64.

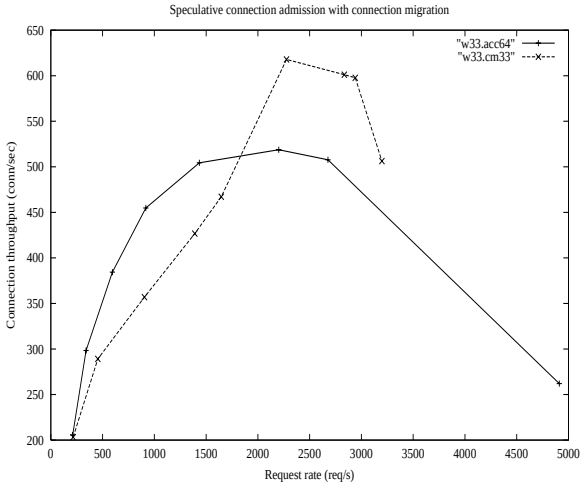


Figure 7.2: Speculative admission within request distribution (The front-end routes 33% of the requests to one server and 67% to the other)

7.2.2 Speculative connection admission in request distribution

Both the TCP connection endpoint migration and the speculative connection admission are meant to be used by request dispatching algorithms. In our second experiment, a server accepts speculatively connections only to migrate them to another server. We report the connection throughput values for the normal processing (i.e., without speculative admission) and for the speculative admission. The normal processing uses a SYN_RCVD queue length of 128 and an accept queue length of 64 (depicted throughout the graphs by the keyword "acc64"). The speculative connection admission uses two different increments of the base accept queue length (64), namely 32 and 64. On the graphs, the corresponding curves can be recognized by the percentage represented by the admission queue in the total accept

queue (for instance, “cm33” means that the speculative admission operates on a 64 + 32 long accept queue, in which the admission queue represents 33% of the total queue).

For the first experiment, that in which two thirds of the requests hit the server A and one third the server B, we used only an admission queue of 32 entries for our speculative admission policy. The results are reported in Figure 7.2 (the curves on the graph are tagged “w33”). In Figure 7.3 we present the results for the experiment in which the front-end routes three quarters of the requests to the server A and one quarter to the server B (curves tagged “w25”).

As a general observation, all the speculative cases seem to cope better with higher request rates than the normal processing case. In particular, the “w25.cm50” case performs remarkably well, by extending the responsiveness of the server to request rates between 3000 and 4000 req/s. The normal processing cases are limited below 3500 req/s (Figure 7.2) and 3000 req/s (Figure 7.3), respectively. At these figures, the server gets saturated and the client running the benchmark exhausts the local resources and cannot create more connections (i.e., there are too many open connections to which the server did not respond yet).

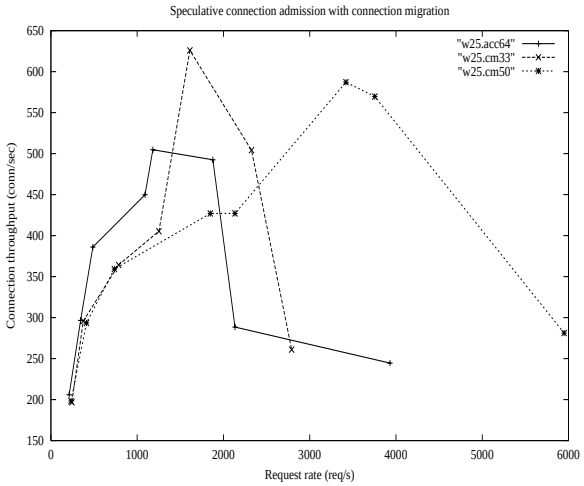


Figure 7.3: Speculative admission within request distribution (The front-end routes 25% of the requests to one server and 75% to the other)

For the case in which the front-end routes two thirds of the requests to one server and the rest to the second server (Figure 7.2), it is worth noting that, for small request rates, the speculative policy induces a significant overhead. For such request rates, the normal processing case outperforms the speculative case. Nevertheless, as soon as the request rates become important (around 2000 req/s), the

server using speculative admission copes better with the increased demands. It yields the peak performance at about 617 conn/s and some 2200 req/s. This trend becomes clear when looking at the second case, where the imbalance is more severe and the speculative admission improves undisputedly the overall performance of the distributed server (see Figure 7.3). Indeed, both speculative cases outperform the normal processing (denoted by "w25.acc64") for high request rates. It is also worth noting that, for high request rates, larger admission queues yield significantly better connection throughput figures. Similar to the first experiment (Figure 7.2), at small request rates, the normal processing handles better the situation, as the overhead of the connection migration cannot be hidden. All these observations are consistent with those made in Section 6.9 for the experiments with the simple connection endpoint migration policy.

Perhaps another aspect worth noting is that the imbalances at the front-end prevent the performance of the two-server node to scale linearly when compared to the single server case. Indeed, by looking at the graphs "single128-64" in Figure 7.1, "w33.acc64" in Figure 7.2 and "w25.acc64" in Figure 7.3, one can see that the peak performance of a single server is at about 300 conn/s (the exact figure is 324 conn/s) while none of the two cases for the two-node server achieves more than 519 conn/s. It is the job of the speculative admission to overcome the imbalance and to improve the performance by almost doubling the peak connection throughput figure.

7.3 Summary

This chapter presented the speculative TCP connection admission, a back-end level mechanism to improve the sub-optimal request distribution decisions in cluster-based servers. The overloaded server nodes in the cluster accept speculatively the incoming requests only to offload them to less-loaded nodes by means of the TCP connection endpoint migration. The speculative connection admission targets back-end level request distribution policies that leverage routing decisions taken outside the cluster. The mechanism has been implemented in the Linux kernel as part of our policy-based software architecture for request distribution. We have been able to show that the speculative connection admission adds little overhead to the normal TCP processing on overloaded nodes, offsets the load imbalances and accommodates high request rates. These results recommend the speculative admission for an active role in the request distribution in cluster-based servers.

Chapter 8

Summary and future work

8.1 Summary

This thesis presented Single System Image (SSI) operating system services for cluster-based Web servers. These services permit an unmodified stand-alone server program (say, Apache [5]) running on a node of a cluster of Commodity Of The Shelf (COTS) computers to regard the entire cluster as a single machine whose resources are at its disposal. Each stand-alone server instance running on a cluster node is unaware of other similar instances running on other cluster nodes. The cooperation among the cluster nodes takes place entirely at the kernel level and is based on cluster-wide SSI services. These services are part of a general software architecture for cluster operating systems based on resource virtualization (see Figure 1.1). Through resource virtualization, the nodes access remote resources as if local.

For the purposes of this thesis, we developed two such SSI services: the Cluster-Aware Remote Disk (CARD) drivers and the TCP connection endpoint migration. The CARD drivers virtualize local accesses of a node to remote disks and provide for an extended, cluster-wide disk cache built by means of cooperative caching. The TCP connection endpoint migration establishes a server equivalence among the nodes of the cluster by using generic server-side endpoints that can be arbitrarily assigned and reassigned to any server machine in the cluster.

8.1.1 CARDs and cooperative caching

The main incentive behind our system design is represented by the performance figures of the System Area Networks (SANs) that are commonly used nowadays to enhance the traditional networking capabilities of the COTS clusters (see Figure 1.2). The disk virtualization methods and the cooperative caching traffic as well

as the TCP connection endpoint migration protocol make use of this high speed interconnect. Therefore, the first endeavors of this thesis were to assess the effectiveness of the resource virtualization techniques based on SANs.

When no cooperative caching policy is in use, the CARD drivers behave like remote disk interfaces on the local machine. We show that the CARD drivers yield good performance due to their highly asynchronous mode of operation: cluster-wide, in terms of overlapping the communication with the computation between the two nodes (the remote disk node and the node mounting the remote disk through a CARD driver), as well as locally, at the remote disk site, through a mix of interrupt-time and thread-based block request processing. This mixed processing method aims to maximize the degree of pseudo-parallelism between the request handling and the data delivery on a uniprocessor machine. These conclusions are validated experimentally in Section 3.11 and support **Claim 2** as stated in Section 1.3.

The CARD drivers offer the upper kernel layers as well as the applications like our SSI server the benefit of using I/O page/block abstractions built on top of the message passing capabilities of the SAN. A fine-grain control over these abstractions is made possible through a single copy protocol that allows for exclusive caching. The implementation of this single copy protocol uses the capability to perform DMA from the disk to the network buffering system without intermediate copying. The description of this mechanism provided in Section 3.7 supports **Claim 3** in Section 1.3.

A further compelling argument for building high level kernel abstractions is that the bottleneck of the computing systems is not in the hardware anymore, but rather in the software. For instance, we have been able to show (see Section 3.8) that exporting the effect of the read-ahead policy of a file system using the CARD drivers is highly sensitive to the inner parameters of the driver implementation (namely the maximum read-ahead window). Our CARD driver delivers an optimal performance for a maximum read-ahead window equal to that of the remote disk system. This fact shows optimal SAN usage. For other values however, even for those that would favor higher degrees of asynchrony (and thus better SAN usage), the performance drops. Put differently, the high performance SAN figures do not translate automatically into high performance CARD figures. In general, the SSI services based on SANs need to harmonize with the operation and the features of the software using the SAN (the operating system kernel being of major concern). This statement, together with the experimental results from Section 3.11 concerning the mixed blocking/event-driven mode of operation of the CARD drivers and the performance of our exclusive caching implementation support **Claim 1**.

The virtual disks share data across the cluster, but the corresponding local disk caches are still managed separately. We used cooperative caching to provide for a decentralized, globally-coordinated SSI disk cache. We designed and

developed a cooperative caching algorithm called Home-based Server-less Cooperative Caching (HSCC) and showed that the cooperative caching algorithms run efficiently only when considering additional information such as the loads of the cluster nodes (see Section 4.7). We proved that, for large workloads, most of the cooperative cache hits are global hits (i.e., remote cache hits) and, therefore, the eviction handling pays off only when choosing carefully the target of the block saving attempt. These results support **Claim 4**.

In the case of the Web systems, which have different workloads than the file systems, we found that the cooperative caching is effective for cluster-based Web servers if supported by application-level hints. More specifically, we used information from a Web request distribution curve to steer at fine-grain level the operation of our general purpose cooperative caching algorithm (HSCC). Supplying that kind of information to the kernel-level code is possible due to the flexible/extensible design of our software architecture that enables the applications to download their policies into the kernel at will, similar to the extensible kernels. Our general purpose algorithm (HSCC) could not fill up the performance gap between a simple solution like replication and one using data sharing through virtual disks (more precisely, our CARD drivers). Only when targeting the classes of the large and unpopular Web requests (i.e., when driven by request distribution-induced hints) HSCC makes up for the performance loss and compares favorably to replication (see Section 5.3). This result supports **Claims 5** and **10**.

8.1.2 TCP connection endpoint migration

Cooperative caching by itself cannot answer all the questions raised by a cluster-based Web server. The main problem arises when trying to reconcile the data locality with the load balancing. Although our HSCC algorithm attempts to even out the loads of the cluster caches, the imbalances in a cluster-based Web server occur also due to request routing decisions and cannot be offset through cooperative caching alone. For that, one needs a request routing mechanism as well. Since the nodes jointly managing the cooperative cache by means of HSCC already share load information, the natural choice is to develop a back-end level request routing mechanism, in this case the TCP connection endpoint migration mechanism.

Through TCP connection endpoint migration, generic server-side connection endpoints can be arbitrarily assigned and reassigned later to server nodes in the cluster. We show that the TCP connection endpoint migration performs well for back-end level request distribution schemes too, not only for fault-tolerant purposes as shown by the previous research on general, client-server TCP connection migration protocols. In the particular case of the cluster-based Web servers, we show that the TCP connection endpoint migration is an effective ingredient of

back-end level request distribution policies, both for persistent and non-persistent HTTP connections.

For instance, simple request distribution policies migrating non-persistent HTTP requests for small and popular static Web documents outperform Round Robin by over 75% in terms of overall figures, while the individual improvements for the targeted classes of documents are as high as 2.79 times those of the corresponding Round Robin figures at no extra cost for the other classes of requests (see Section 6.10). The case of migrating persistent HTTP connections is more sensitive to the caching effectiveness of the policy in use, but our experimental results (see Section 6.10.3) show that the TCP connection endpoint migration performs well in this case, too. Thus, the TCP connection endpoint migration can be thought of as an effective back-end level connection routing mechanism for cluster-based servers and may turn out to be the right companion of cooperative caching in the locality-aware request routing policies attempting to trade off between data and connection migration. This statement validates our **Claim 6** in Section 1.3.

We also developed a back-end level load balancing mechanism based on the TCP connection endpoint migration that we call speculative TCP connection admission. This mechanism attempts to alleviate the consequences of the sub-optimal routing decisions taken outside the cluster-based server (either by the front-ends or by an external entity routing the requests as it is the case with the Round Robin DNS or with the multi-tier server architectures). This goal is accomplished by offsetting the induced load imbalances late, at service time (that is to say, when the back-end servers can inspect the request content). By enlarging the accept queue of the individual servers in the cluster, an overloaded server node can accommodate an increased number of incoming requests only to offload them onto lighter-loaded nodes by means of the TCP connection endpoint migration. Our experimental results (see Section 7.2) show that the speculative TCP connection admission effectively offsets the externally induced imbalances among the cluster nodes and thus validate **Claim 7** in Section 1.3.

8.1.3 Kernel code development, experience and benefits

The kernel code development is not an easy task. It involves a specific type of programming and, most of the time, it lacks the support of comfortable development tools. When we say specific programming techniques we refer to the fact that the kernel acts as a collection of coroutines, and writing kernel code comes down to writing such code. More than that, even on uniprocessor machines, some of the kernel code is prone to race conditions and needs the appropriate concurrent programming techniques. This is the case of the software and hardware interrupt handlers that run when the interrupts stop the current thread of execution in the

kernel. And above all, the kernel address space is a shared one, and the developer cannot indulge in relying on protection domains as in the case of the user-level development. All the common programming mistakes (unallocated memory, memory leaks, writing beyond the bounds of the allocated memory, etc.) end up by freezing the machine. Last but not least, the complexity of the operating system, the largest software run by any machine, may seem daunting at times. Getting accustomed to kernel software is a process whose learning curve has a slow-increasing slope. All these problems make the development harder and more time-consuming.

Nevertheless, we considered it quintessential to implement our SSI services at kernel level. First of all, conceptually, these services belong to the operating system as they provide basic functionality to the upper kernel layers and applications. Moreover, our kernel services offer simplicity of use and flexibility in operation. For instance, remote disks can be mounted locally by means of the CARD drivers as any local disk would be (that is, by using the regular Unix *mount* command). The applications can download at will their favorite cooperative caching policy into the CARD driver and thus they steer the global management of the cluster-wide cooperative cache (see Chapter 4). These statements support **Claim 8** in Section 1.3.

Equally simple, an application developer willing to write a request distribution policy based on the TCP connection endpoint migration needs to specify only the migration target and the heuristics triggering the migration (see Section 6.8.1) and to download them into the kernel. The generic TCP connection endpoints identify the cluster as a whole, rather than individual server nodes, and establish a logical equivalence of all the servers in the cluster. These arguments provide further support for **Claim 8** in Section 1.3.

A kernel implementation of the SSI services enables unmodified stand-alone applications (the Apache server in the case of our SSI cluster-based Web server, see Chapter 5) to run in a distributed/parallel environment. Our solution validates **Claim 9** in Section 1.3. The alternative would be to implement similar services in the user space. This solution not only implies application changes, but is also known to complicate the software development and to induce performance penalties as the underlying stand-alone kernel software is unaware of the distributed/parallel environment. Therefore, its inner mechanisms and policies fail to match the expectations of the applications.

However, the kernels have been accused of adapting their services too slowly to the new emerging paradigms of computation. A central point of criticism was related to the general purpose operating system algorithms that were supposed to fulfill every application need with high performance and yet they failed to do so. With the advent of the Internet and multimedia applications, it became clear that the flexibility/extensibility in the kernel is a must. Exo-kernels and grafting kernels provide such flexibility. Faithful to this perspective, we designed flexible SSI

service software in a policy-oriented (server) architecture that allows applications (server programs) expressing their views through policies. These policies can be downloaded into the kernel at will. In turn, the kernel honors the application's view by running the policy code in kernel context. This mechanism enabled experiments with various cooperative caching and request distribution policies. We have been able to show that general purpose algorithms are outperformed by algorithms that take into account the application specificity (see the case of the request distribution-aware caching described in Chapter 5 or the comparisons between the request distribution policies based on the TCP connection endpoint migration and Round Robin in Chapter 6). These results validate our last claim (see **Claim 10** in Section 1.3).

8.2 Future work

There is definitely a lot of work to do in the area of the SSI cluster-based servers. However, we would like to restrict this final part of the thesis to a few issues that we consider of immediate relevance to what we have presented so far.

8.2.1 The scalability analysis of HSCC and the locality-aware request distribution policies using the TCP connection endpoint migration

In Chapter 4, when we discussed the scalability of our Home-based Server-less Cooperative Caching algorithm, we mentioned that we couldn't test its performance on large scale clusters. A theoretic discussion in Subsection 4.5 supports the scalability of the HSCC lookup and eviction handling procedures. However, a given home node can have a hard time coping with situations in which all the cluster nodes look up simultaneously blocks managed by that home. The situation may be worsened as the protocol runs in bottom halfs at interrupt time and can create the conditions for a receiver live-lock (the system functions, but spends all of its processing time by serving network interrupts). Before suggesting any solutions for this kind of possible shortcomings of our algorithm, we would like to develop methods and tools to analyze this type of receiver live-lock in order to assess its impact on the overall performance of the algorithm.

A further point of interest concerns the scalability of the locality-aware request distribution policies using the TCP connection endpoint migration. Assessing the TCP connection endpoint migration overhead for large scale clusters is an important issue. We believe that distributing this overhead over several back-end servers like we did in Subsection 6.10.2 may be the starting point of a future investigation.

Naturally, this issue has to be correlated with those presented further in Subsection 8.2.3.

8.2.2 Cluster provisioning

A serious problem concerns the cluster provisioning. Choosing the right cluster size in terms of resources can prevent worst case scenarios like that described in the previous subsection from happening. In general, the ability to estimate the resource needs of a certain workload may bring benefits beyond that of reaching the optimal performance by setting clear bounds on the scalability (and thus, by pointing out the thresholds on the usability of the cluster). A thorough analysis of the resource needs of a cluster-based server may lead to a better cluster usage by restricting the number of the nodes allocated for a given task.

8.2.3 Locality-aware request distribution policies

The next logical step when having the cooperative caching and the TCP connection endpoint migration at one's disposal is to mix the two in locality-aware request distribution policies. As mentioned in Chapter 4, when we presented our HSCC algorithm, the homes maintain references to the nodes that hold the last recently loaded blocks for which the home takes the responsibility. That means that the knowledge about the caching of a given file is spread throughout the cluster. A request hitting a node would then have to figure out for which of the blocks of the requested file the node is a home. Then, by inspecting the references that the home has for those blocks, a locality-aware request distribution policy would be able to exploit the locality of reference by migrating the request to a node that caches the requested file.

In fact, there is more at stake than the locality of reference for the requested data. As we already mentioned in Section 2.1.4, reconciling the load balancing with the locality of reference is a hard task as the two goals are opposite: routing all the requests to the same node in order to maximize the data locality results in hot-spots, while striving to achieve good load balancing entails spreading multiple copies of a file over several nodes in the cluster, which represents both a waste of memory and an increased network and/or disk I/O activity.

One can use the cooperative caching and the TCP connection endpoint migration to attempt to reach a heuristic optimum by switching between data and request migration. Remember that HSCC maintains on every node an approximate evaluation of the load of the other nodes in terms of the blocks stored on behalf of the other nodes. This information is used to steer the operation of the eviction handling procedures by avoiding to send the evicted blocks to overloaded nodes. This type

of information flows naturally across the cluster through the HSCC algorithm and can help reach connection endpoint migration decisions as well.

For instance, knowing that an overloaded node caches a copy of a requested file may prevent migrating the request to that node as it may very well happen that that node will evict exactly the copy of that file (or might have already done so) before the corresponding TCP connection endpoint gets migrated there. Moreover, even if an optimistic policy might choose to migrate the request under these circumstances, the request service may face other problems, even though it might count on data locality. Such problems include the overheads due to the memory swapping and the poor network subsystem performance due to the scarcely available memory.

On the other hand, knowing that there are lightly loaded nodes available may be an incentive to migrate requests, even if the data locality is missing, in order to balance the cluster-based server.

Bibliography

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)* , October 1998.

[2] Woo Hyun Ahn, Woo Jin Kim, and Daeyon Park. Content-Aware Cooperative Caching for Cluster-Based Web Servers. In *The Elsevier Journal of Systems and Software 69 (2004), pag. 75-86*, 2004.

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. In *IEEE Computer, Vol. 29, No. 2, pp. 18-28*, February 1996.

[4] T. Anderson, M. Dahlin, J. M. Neefe, D. Patterson, D. Rosseli, and R. Y. Wang. Serverless Network File Systems. In *The 15th Symposium on Operating System Principles*, December 1995.

[5] Apache. *http:/www.apache.org/*.

[6] VIA: The Virtual Interface Architecture. *http://www.viarch.org*, 1998.

[7] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient Support for P-HTTP in Cluster-Based Web Servers. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

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

[9] Maurice J. Bach. *The Design of the Unix Operating System*, 1986. Prentice Hall Inc.

[10] G. Banga and P. Druschel. Measuring the capacity of a Web Server. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems*, December 1997.

[11] G. Banga, P. Druschel, and J. Mogul. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, October 1996.

[12] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems . In *Proceedings of the Third Symposium on Operating System Design and Implementation* , February 1999.

[13] T. Berners-Lee, R. Fielding, and H. Frystyk. *RFC 1945: Hypertext Transfer Protocol – HTTP 1.0*, 1996.

[14] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, December 1995.

[15] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom '99*, March 1999.

[16] T. Brisco. *RFC 1764: DNS Support for Load Balanacing*, 1995.

[17] R. B. Bunt, D. L. Eager, G. M. Oster, and C. L. Williamson. Achieving Load Balance and Effective Caching in Clustered Web Servers. In *Proceedings of the Fourth International Web Caching Workshop*, March 1999.

[18] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The State of the Art in Locally Distributed Web-Server Systems. In *ACM Computing Surveys, Vol. 34, No.2, pp. 263-311*, June 2002.

[19] Enrique V. Carera and Ricardo Bianchini. Improving Disk Throughput in Data-Intensive Servers. In *Proceedings of the 10th IEEE International Symposium on High Performance Computer Architecture*, February 2004.

[20] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A parallel file I/O interface for MPI. Technical Report NAS-95-002, NASA Ames Research Center, Moffet Field, CA, January 1995.

[21] T. Cortes, S. Girona, and L. Labarta. PACA: A Distributed File System Cache for Parallel Machines. Performance under Unix-like workload. Technical Report UPC-DAC-RR-95/20 or UPC-CEPBA-RR-95/13, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, 1995.

[22] M. E. Crovella, R. Frangioso, and M. Harchol-Balter. Connection Scheduling in Web Servers. In *Proceedings of the 2nd Usenix Symposium on Internet Technologies and Systems*, October 1999.

[23] M. Dahlin, R. Yang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *The First Symposium on Operating Systems Design and Implementation*, November 1994.

[24] O. P. Damani, P. E. Chung, Y. Huang, , C. Kintala, and Y-M. Wang. ONE-IP: Techniques for Hosting a Service on a Cluster of Machines. In *Proceedings of the 6th International WWW Conference*, April 1997.

[25] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthut Lent, Dave Noveck, Tom Talpey, and Mark Wittle. The Direct Access File System. In *Proceedings of the 2nd Usenix Conference on File and Storage Technologies (FAST '03)*, pages 175–188, San Francisco, USA, March–April 2003.

[26] J. del Rosatio, R. Bordawekar, and A. Choudhary. Improve parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS Workshop on Input/Output in Parallel Computer Systems*, 1993.

[27] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Proceedings of Hot Interconnects V*, August 1997.

[28] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[29] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, and Henry M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.

[30] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *RFC 2068: Hypertext Transfer Protocol – HTTP 1.1*, 1997.

[31] I. Foster, Jr. D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast Access to Distant Storage. In *Proceedings of the Fifth Annual Workshop on I/O in Parallel and Distributed Systems*, November 1997.

[32] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, July 1997.

[33] S. Glassman. A Caching Relay for the World Wide Web. In *First International World Wide Web Conference*, 1994.

[34] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-Based Scheduling to Improve Web Performance. In *ACM Transactions on Computer Systems, Vol. 21, No.2, pp. 207-233*, May 2003.

[35] J. H. Hartman, I. Murdock, and T. Spalink. The Swarm Scalable Storage System. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, June 1999.

[36] R. S. C. Ho, K. Hwang, and H. Jin. Single I/O space for Scalable Cluster Computing. In *Proceedings of the 1st IEEE International Workshop on Cluster Computing*, December 1999.

[37] G.D.H. Hunt, G.S. Goldszmidt, R.P. King, , and R. Mukherjee. Network Dispatcher: a connection router for scalable Internet services. In *Proceedings of the 7th International WWW Conference*, April 1998.

[38] W. B. Ligon III and R. B. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, June 1999.

[39] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and W. Tichy. Integrating collective I/O and cooperative caching into the Clusterfile parallel file system. In *Proceedings of the 18th ACM International Conference on Supercomputing (ICS)*, June 2004.

[40] F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. In *Third IEEE International Conference on Cluster Computing*, October 2001.

[41] Editor J. Postel. *RFC 793: Transmission Control Protocol Specification*, 1981.

[42] V. Jacobson, R. Braden, and D. Borman. *RFC 1072: TCP Extensions for High Performance*, 1992.

[43] Kangho Kim, Jin-Soo Kim, and Sungin Jung. A Network Block Device Over Virtual Interface Architecture on LINUX. In *Proceedings of the IEEE International Parallel and Distributed Symposium*, April 2002.

[44] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[45] David Kotz and Ravi Jain. I/O in parallel and distributed systems. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 40, pages 141–154. Marcel Dekker, Inc., 1999. Supplement 25.

[46] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[47] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best. File Access Characteristics of Parallel Scientific Workloads. In *IEEE Transactions on Parallel and Distributed Systems, 7(10)*, October 1996.

[48] V. Olaru and W. F. Tichy. CARDs: Cluster Aware Remote Disks. In *Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, May 2003.

[49] V. Olaru and W. F. Tichy. On the Design and Performance of Remote Disk Drivers for Clusters of PCs. In *Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'04)*, June 2004.

[50] V. Olaru and W. F. Tichy. Request Distribution-Aware Caching in Cluster-Based Web Servers. In *Proceedings of the Third IEEE International Symposium on Network Computing and Applications (IEEE NCA04)*, August 2004.

[51] V. Olaru and W. F. Tichy. Speculative TCP Connection Admission using Connection Migration in Cluster-based Servers. In *Proceedings of the IADIS International Conference WWW/Internet 2004*, October 2004.

[52] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network

Servers. In *Proceedings of the ACM Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)* , October 1998.

[53] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: An Unified I/O Buffering and Caching System . In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[54] S. Gadde R. P. Doyle, J. S. Chase and A. Vahdat. The Trickle-Down Effect: Web Caching and Server Request Distribution. In *Proceedings of Sixth International Workshop on Web Caching and Content Distribution (WCW'01)*, June 2001.

[55] Prasenjit Sarkar and John H. Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[56] Huseyin Simitici and Daniel A. Reed. A Comparison of Logical and Physical Parallel I/O Patterns. In *International Journal of High Performance Computing Applications, special issue (I/O in Parallel Applications), 12(3)*, 1998.

[57] M. Sivathanu, V. Prabhakaran, F. Popovici, T. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd Usenix Conference on File and Storage Technologies (FAST '03)*, March 2003.

[58] Evgenia Smirni and Daniel A. Reed. Workload Characterization of I/O Intensive Par allel Applications. In *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation,Springer-Verlag Lecture Notes in Computer Science*, June 1997.

[59] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *Proceedings of the 6th ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '00)*, August 2000.

[60] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-Grained Failover Using Connection Migration. In *Proceedings of the Third Annual USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

[61] The Standard Performance Evaluation Corporation (SPEC). *http://www.spec.org/web99/*.

[62] W. Richard Stevens. *TCP/IP Illustrated, Volume 1*, 1994. Addison Wesley Longman, Inc.

[63] F. Sultan, A. Bohra, and L. Iftode. Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions. In *Proceeding of the 22nd Symposium on Reliable Distributed Systems (SRDS)*, July 2003.

[64] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly Available Internet Services Using Connection Migration. Technical Report DCS-TR-462, Department of Computer Science, Rutgers University, Piscataway, NJ 08854-8019, December 2001.

[65] Andrew S. Tanenbaum. *Distributed Operating Systems*, 1995. Prentice Hall Inc.

[66] Myricom Inc. GM: the low-level message-passing system for Myrinet networks. *http://www.myri.com/scs/index.html*.

[67] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, 1997.

[68] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the ACM Symposium on Operating Systems Principles*, December 1995.

[69] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[70] T. M. Warschko, J. M. Blum, and W. F. Tichy. On the Design and Semantics of User-Space Communication Subsystems . In *Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'99)*, June 1999.

[71] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, December 1999.

[72] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference*, pages 161–175, 2002.

[73] C.-S. Yang and M.-Y. Luo. Efficient Support for Content-Based Routing in Web Server Clusters. In *Proceedings of the 2nd Usenix Symposium on Internet Technologies and Systems*, October 1999.

[74] K. G. Yocum, D. C. Anderson, J. S. Chase, and A. Vahdat. Anypoint: Extensible Transport Switching on the Edge. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[75] X. Zhang, M. Barrientos, J.Chen, and M. Seltzer. HACC: An Architecture for Cluster-Based Web Servers. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[76] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[77] G. Zipf. *Human Behavior and the Principle of Least Effort*, 1949. Addison Wesley.

List of abbreviations

CARD	Cluster-Aware Remote Disk, 17
COTS	Commodity-Off-The-Shelf (clusters), 13
DAFS	Direct Access File System, 38
DSM	Distributed Shared Memory, 14
GMS	Global Memory System, 62
HDC	Hash Distributed Caching, 72
HSCC	Home-based Server-less Cooperative Caching, 17
ISN	Initial Sequence Number, 97
LAN	Local Area Network, 28
LARD	Locality-Aware Request Distribution, 27
MAC	Medium Access Control, 25
NASD	Network-Attached Secure Disks, 38
RDMA	Remote Direct Memory Access, 38
SAN	System Area Network, 13
SSI	Single System Image, 13
VIA	Virtual Interface Architecture, 39
VIP	Virtual IP (address), 24