On the Design and Performance of Kernel-level TCP Connection Endpoint Migration in Cluster-Based Servers

Vlad Olaru Walter F. Tichy

Computer Science Department
Karlsruhe University , Germany
E-mail: {olaru,tichy}@ipd.uka.de

Abstract

The TCP connection endpoint migration allows arbitrary server-side connection endpoint assignments to server nodes in cluster-based servers. The mechanism is client-transparent and supports back-end level request dispatching. It has been implemented in the Linux kernel and can be used as part of a policy-based software architecture for request distribution. We show that the TCP connection endpoint migration can be successfully used for request distribution in cluster-based Web servers, both for persistent and non-persistent HTTP connections. We present locality-aware policies using TCP connection migration that outperform Round Robin by factors as high as 2.79 in terms of the average response time for certain classes of requests.

1 Introduction

Cluster-based servers are common place nowadays in the server industry. They employ front-end computers to redirect the client requests to server computers (back-ends). The requests are most of the time dispatched according to back-end load balancing and/or content-aware hints. However, previous research results [9, 2] suggest moving the request distribution at the back-end level in order to take advantage of the back-end cooperation.

In this paper we describe the TCP connection endpoint migration, a mechanism that helps build back-end level request routing policies by allowing a flexible assignment and reassignment of 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.

To verify our mechanism, we used it in content-aware request distribution policies both for persistent and non-persistent HTTP connections for a popular stand-alone Web server, namely Apache [1]. We run Apache transparently on top of our cluster (one server instance per cluster node), without any modifications to the server code. Only the underlying kernels are aware of the inter-node cooperation, while the user-level daemon serves requests as it would do on a single machine.

2 Related work

Connection migration is a new request routing mechanism lately put under scrutiny by Snoeren et al. [10, 11] and Sultan et al. [15, 14]. 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 true TCP-migration protocol). Both solutions describe client-server migration protocols allowing either one of the parties a graceful migration of their corresponding endpoint to a third party conforming to the protocol. No front-ends are needed between the client and the server.

However, there is little evidence that the client-server connection migration could be used successfully in request distribution for cluster servers mostly because of the induced 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 [11]. In a somewhat different domain, they also used the connection migration to approach host mobility [10] (because the client endpoints can migrate as well). With Server Continuations, Sultan et al. [14] used the connection migration to migrate server sessions (a particular form of process/thread migration). Our protocol is client-transparent and allows server-side endpoint migrations only, in order to fully profit from the high speed System Area Networks, whose latency and throughput figures are better than those of the Internet.

A flexible and general one-to-many communication model is offered by the Anypoint [17] system. Anypoint uses 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.

Cardellini et al. [5] provide a broad survey on locally distributed Web servers. The survey doesn't mention connection migration among the request routing mechanisms.

3 Server and request distribution architecture

Our server relies on the capabilities of a System Area Network (SAN) acting as a communication backplane among the cluster nodes. The intra-cluster specific protocols take place on the SAN, while the traditional communication to the "world" employs the LAN. The low latency and high bandwidth figures of the SANs represent the incentive for our connection endpoint migration protocol design.

Following the taxonomy proposed by Cardellini et al. [5], our architecture employs features of both *Cluster-based* and *Virtual servers*. Similar to *Cluster-based servers*, the front-ends may route requests as *TCP-layer* switches. However, similar to *Virtual servers*, a *Virtual IP* server address is assigned to each of the back-ends and not to the front-end(s). A (*Virtual IP, service port*) pair defines a *generic* TCP endpoint. The connections linked to this generic endpoint can migrate from one server node to another through the connection endpoint migration mechanism.

The central point of the server software architecture is a three-phase request distribution algorithm. One phase is executed at the front-ends, while the other two take place at the back-end level and express user-specified policies. The design aims at a tight collaboration between the applications and the underlying kernel. Capitalizing on the extensible/grafting kernels experience [4, 6], our design enables the applications (in this particular case, the server programs) to specify their own policies and to download them into the kernel as phases in the request distribution algorithm. Our request distribution is *policy-* and not *service-oriented*.

The first phase performs a blind (non-informed) distribution at the front-end. The distribution is hash-based in order to dispatch the requests uniformly and is deliberately simple in order to offload the front-end. This first phase may be circumvented if using a void policy. As a result, the connection routing is completely disregarded by the front-end(s).

The second phase concerns the load balancing and takes place at the back-end level at the connection setup time. The incoming SYN packets that target generic TCP endpoints may trigger a TCP and CPU load check on that machine. If the machine is heavily loaded, the SYN packet is redirected to a lighter loaded node. The cost of the extra hop is the SAN latency of a SYN packet, amortized over the cost of the entire connection activity. This cost is negligible as it happens only once at the connection setup time.

The third request distribution phase takes place at a back-end machine already connected to the client, either as a result of a front-end or a back-end redirection. This phase is entirely policy-driven and may cause connection endpoint migrations. The normal application-level protocol (HTTP, for instance) processing is carried out and steered through hints provided by the distribution policy. A typical example for the third phase is the content-aware routing.

4 TCP connection endpoint migration overview

Our TCP connection endpoint migration is client-transparent and targets locally-distributed server architectures. It has two variants, one involving the front-end(s) and a fully-distributed one. As a connection request (a SYN packet, in fact) arrives at a front-end, it can be directed to a given server according to a certain policy or it can pass through to eventually hit 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 the 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, among other information, 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, the requests reach back-ends without front-end interference. If the 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. The migration leaves no connection behind, so all the subsequent packets of the migrated connection will be delivered to the usual RST mechanism of TCP [13, 8]. The routine responsible to send back an RST segment to the client looks up an entry corresponding to the packet in the forwarding table. If found, no RST segment will be generated and, instead of discarding the packet, it is sent further over the SAN according to the information stored in the forwarding table. 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 forwarding 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 and the new server fulfill a modified version of the normal TCP connection setup protocol [8]. As a result, 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 truly becomes a duplicate of the old server's endpoint when the initiating back-end

transfers the entire connection endpoint status to the new site. 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.

5 The connection endpoint migration protocol

The operation of a server wishing to migrate one of its connection endpoints is described in Figure 1. As soon as a migration request is issued (i.e., a migration SYN is sent to the remote server), the connection moves to a wait state. While in the wait state, any incoming packet is stored in a connection checkpoint.

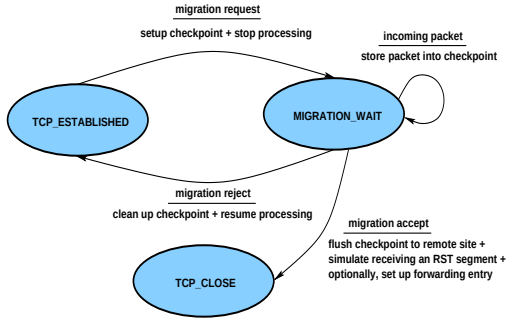

Figure 1. Connection migration operation at the initiator

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) [8]. 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 migration requester has to choose a proper "client" ISN. Additionally, it also has to impose the ISN that the target node will use as its own ISN during the connection setup protocol. Letting the target server choose its own ISN would not permit matching the server sequence numbers currently in use. The ISN of the new server is the next allowed send sequence number (denoted by *snd_nxt* in RFC 793 [8]) for the old server.

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 [8]), minus one. With the ISNs chosen like that, the modified version of the TCP connection setup protocol looks like this:

- The initiator server sends to the new server a modified SYN packet that carries the "client" and the "imposed" ISNs presented before. At the remote TCP stack, the imposed ISN is adopted 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 and issues locally the corresponding ACK.

As it can be noticed, only one message is requested to set up a new connection endpoint at the new site. This design saves us one SAN message (namely the final ACK, since 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 the request distribution.

When the new endpoint is fully set up at the remote site, a *migration acknowledgment* is sent back to the old server. 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. For the back-end version, it also sets up the appropriate forwarding entry. The remote site can safely replay the checkpoint because of the properly set sequence numbers. In turn, the server program carries out the application-level protocol. The machine that initiated the migration simulates receiving an RST segment (see [8, 13]) and thus flushes its connection state and allocated resources.

The connection migration checkpoint is built at the connection setup time and stores incoming packets. It is periodically cleaned up as the request processing is carried out. It isolates the server from the client involvement during the migration. During the replay of the checkpoint at the remote site, the automatically generated ACKs of the TCP engine are dropped locally for performance rather than correctness reasons. Since some of the checkpoint packets have been acknowledged once by the old server, the new server doesn't want to confuse the client by sending duplicate ACKs as these may trigger on the client side the congestion avoidance [13] algorithm. As a result of that, the future data transfers will be erroneously penalized in terms of performance, since, actually, no congestion took place. Also, during the migration process, the node initiating the migration suppresses all the packets that may be sent to the client (including ACKs sent in response to the client packets received after initiating the migration).

For the front-end version, as soon as the migration completes, each ACK sent by the new server is used to update the corresponding front-end connection routing entry, if any. If no entry for the connection exists, the front-end registers the connection-to-Medium Access Control address mapping of the outgoing packet. As soon as the first client ACK arrives at the new server, the updating process stops (i.e., server ACKs act no more as update messages).

Upon migration failure, a fall-back mechanism resumes the execution at the old server. A *migration reject* message is sent back and recognized by the old server as an error

condition. Normally, the request processing is continued from where it was left, if not decided otherwise by the the server policy that triggered the migration. Anyway, as soon as the migration rejection is recognized, the connection is again viewed as a regular, locally-bound connection.

Finally, a few words on time synchronization. TCP uses time stamps to prevent either peer to receive stale packets. These time stamps are usually taken from the local logical clock (i.e., an integer counter) of the operating system (in Linux, the so called *jiffies*). In a cluster, there is no way to synchronize these internal counters because they have internal relevance only. When it comes to migration, if the old server uses "newer" time stamps than the new server (i.e., the logical clock of the old server is greater than that of the new one), the client TCP engine will refuse the packets of the new server because they have "older" time stamps (i.e., smaller than those expected). Our solution piggybacks the old server's clock value on the migration SYN by using a SYN option [8]. The new server stores the value in the environment of the migrated connection. A filter installed on the outgoing TCP kernel path uses then this value to adjust the time stamps accordingly in every outgoing TCP segment.

6 Putting the policies and the connection endpoint migration to work

Technically speaking, writing a policy using connection endpoint migration is simply a matter of changing the behavior of the kernel routine that handles the socket queue which stores the incoming packets. This queue is part of the checkpoint managed by the migratory connections. The server applications access the data in this queue through the *read/readv* or *recv/recvfrom/recvmsg* system calls. Inside the Linux kernel, these routines call *tcp_recvmsg* which checks the receive queue and transfers the data to the user space. The policy operates inside this method by avoiding the data transfer to the local server in order to migrate the connection endpoint.

We wrote two such policies, both locality-aware. They exploit the knowledge gained by inspecting a Web request distribution curve respecting a Zipf-like [18] law. The idea is to identify classes of requests and to take advantage of the data locality by migrating all the requests of one class to the same server. As an example, we use for our experiments the WebStone [12] data set and its default mix of request classes: class0 (35% of the total number of requests), class1 (50%), class2 (14%) and class3 (1%).

7 Performance evaluation

The WebStone data set is replicated on local disks on each back-end server. The general idea of the experiments is depicted in Figure 2. 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 and decides whether to serve them locally or to migrate them to the server B. In turn, B serves the requests. The performance is always compared to that of a similar server whose front-end dispatches the requests according to a Round Robin policy.

7.1 Experimental setup

The back-end servers are 350 MHz Pentium II PCs with 256 MB of RAM. They run Linux 2.2.14 and are interconnected through a Myrinet switch and LANai 7 cards. The Myrinet cards have a 133 MHz processor on board. They achieve 2 Gb/sec in each direction. The host interface is a 64 bit/66 MHz PCI that can sustain a throughput of 500 MB/sec. The Myrinet cards are controlled by the GM 1.6.4 driver of Myricom [16]. Each back-end runs Apache 1.3.28 [1] as a Web server.

The client 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).

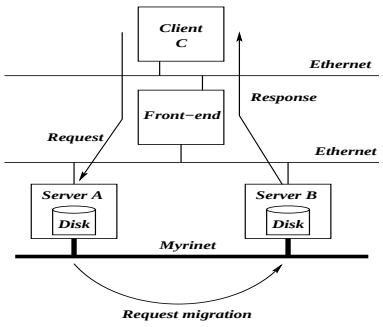

Figure 2. Experimental setup for connection migration policies migrating requests from one back-end server to another

7.2 Non-persistent HTTP connections evaluation

In a first round of experiments, our WebStone client issues plain HTTP 1.0 [3] requests without using the "Keep alive" feature of the protocol (see [7]) in order to ensure that only one request is passed along any given TCP connection. Otherwise said, we did not take advantage of persistent connections to the server. This choice allows observing the caching behavior of a two-node server using a policy that migrates requests for the classes 0 and 1 (small and popular documents) from the server A to the server B like in Figure 2. The overall results are presented in terms of average response time and throughput in Table 1 (lower figures show better response times, higher figures express better throughput). A more refined analysis of the results is possible by having a look at Figure 3, which breaks down the overall

# 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	1615.0
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 1. WebStone overall performance figures for migrating non-persistent HTTP requests for small, popular Web documents (classes 0 and 1) vs. Round Robin

average response time figure into the corresponding class figures.

Figure 3 shows that reassigning the service of the class 0 and 1 documents through connection endpoint migration improves the average response time for these classes. Indeed, for class 0, for instance, the average response time of the connection migration policy experiences reductions of 35.69%, 43.77% and 39.28%, respectively, of the Round Robin average response time. For class 1, the corresponding improvements are smaller (8.55%, 16.43% and 18.56%, respectively).

All these benefits come practically at no extra servicing cost for the classes of large and unpopular files (class 2 and 3). In fact, the figures for class 2 show that the connection migration policy slightly outperforms Round Robin. For class 3, one can notice an insignificant degradation of the response time figure, but no more than 0.03% of the corresponding connection migration figure.

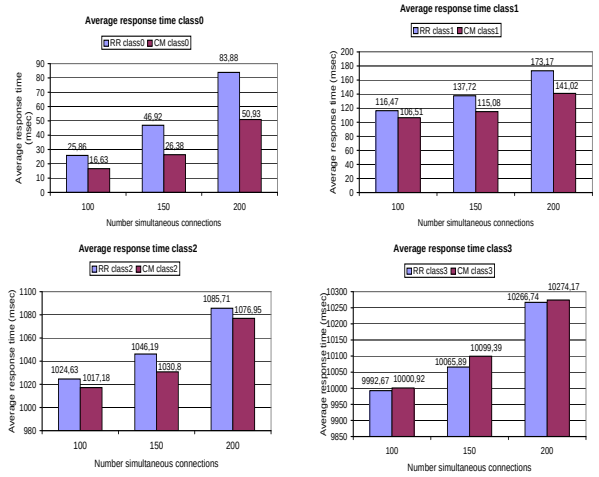

Figure 3. Average class response times for WebStone non-persistent HTTP requests

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 1. Indeed, the overall improvements in the average response time amount to 2.93%, 6.25% and 7.69%, respectively.

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 the Table 1. In order to pinpoint the problem, we decided to scale up the server setup to three back-end servers and to investigate whether the server A (the one initiating the migrations) doesn't become a bottleneck of the system as the load increases.

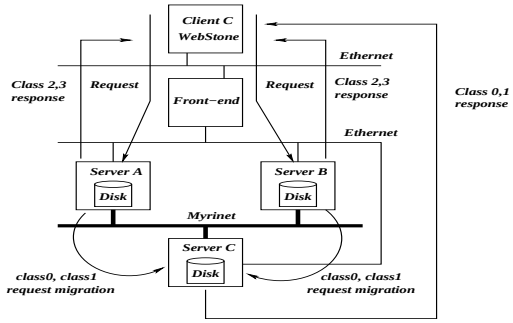

Figure 4. Connection migration policy with three servers

7.3 Non-persistent HTTP connections evaluation of 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 the class 0 and 1 requests to a third back-end server. A visual description of the setup is provided in Figure 4. 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 performance of the cluster-based server. All the experimental results concerning this policy are then compared to those of a policy that dispatches requests to three severs on a Round Robin basis. The comparison of the two policies can be seen in Table 2 and Figure 5.

By looking at these figures, one can infer that, indeed, the problem of the previous connection migration policy was that the server migrating the connections couldn't cope with the overhead of the connection endpoint migration for large workloads. By distributing this overhead almost equally over two servers, one gets a policy that performs again significantly better than Round Robin.

The differences are not significant for 250 simultaneous connections; for this load, the connection migration policy does slightly better than Round Robin. However, as soon

as the load increases to 300 and 350 simultaneous connections, the connection migration policy clearly outperforms Round Robin and copes better with the increased load. A look at Figure 5 proves that, for 300 connections, the class 0 average response time of the connection migration policy is 2.79 times smaller than the corresponding Round Robin time, while for the class 1 the ratio is "only" 2.25. Even the class 2 requests benefit significantly as the average response time for the connection migration policy is 22.79% smaller than that of Round Robin (see Figure 5). For class 3, the gain is practically insignificant, 2.93%.

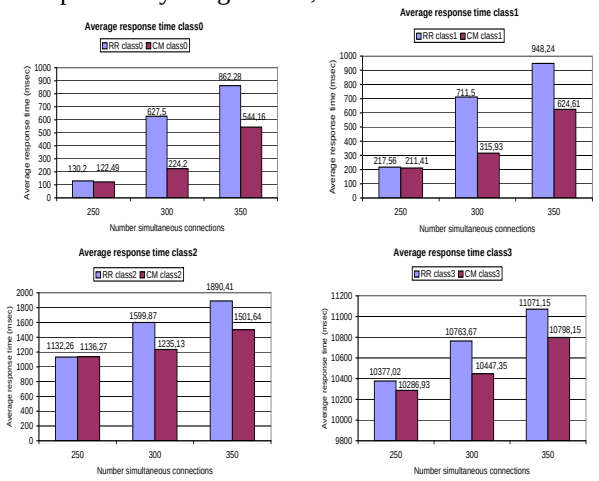

Figure 5. Average class response times for WebStone non-persistent HTTP requests in a three-node cluster-based server

# 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 2. Overall WebStone performance for three servers when migrating non-persistent HTTP requests

A look at Table 2 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, although both of the 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 migration policy outperforms the Round Robin time by a factor of 1.40.

7.4 Persistent HTTP connections evaluation

All the experiments described in the previous two subsections considered non-persistent HTTP connections. Under these circumstances, it is easy to reason about their caching effects. A fair question asks however to assess also TCP connection endpoint migration's performance for persistent connections. Unfortunately, the answer is hardly foreseeable for simple policies like those used before. As soon as the client passes multiple HTTP requests along a given TCP connection, it is almost impossible to guarantee any caching effectiveness for the requested documents. The requests can target different classes and thus pollute the cache of the node where the connection endpoint migrates.

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. 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 by 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 the caching as a result of a connection endpoint migration in the case of the persistent connections. We consider this approach sufficient for the purposes of this paper, which aims to prove only that the connection endpoint migration performs well for persistent HTTP connections too.

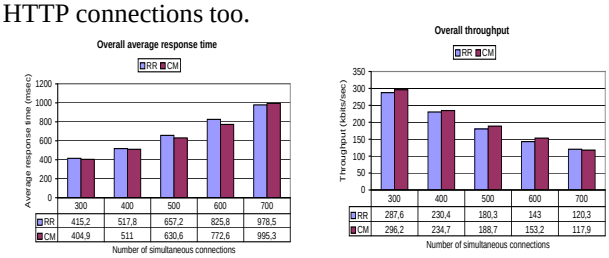

Figure 6. Overall performance figures for WebStone persistent HTTP requests

We use the policy described in Figure 2 to migrate class 2 requests (large and unpopular documents, 14% of the total WebStone requests). The policy operates on a two-node cluster-based server by migrating connection endpoints from one server node to the other one. The overall results are presented in Figure 6. Figure 7 presents class-based breakdowns that help understand easier the caching behavior of the server nodes for particular classes of the requested documents in terms of average response time.

Figure 6 shows that migrating class 2 requests balances the two-node cluster as good as Round Robin does. Indeed, the overall performance figures for the connection endpoint migration 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 Figure 7. For instance, the class 2 graphs show a slight performance degradation due to the connection endpoint migration over-

head. The highest degradation concerns the 700 simultaneous connections case. The penalty is an extra 12.88% of the class 2 Round Robin average response time. In fact, for 700 simultaneous connections, Round Robin outperforms the connection endpoint migration policy for class 0 and class 1 documents as well.

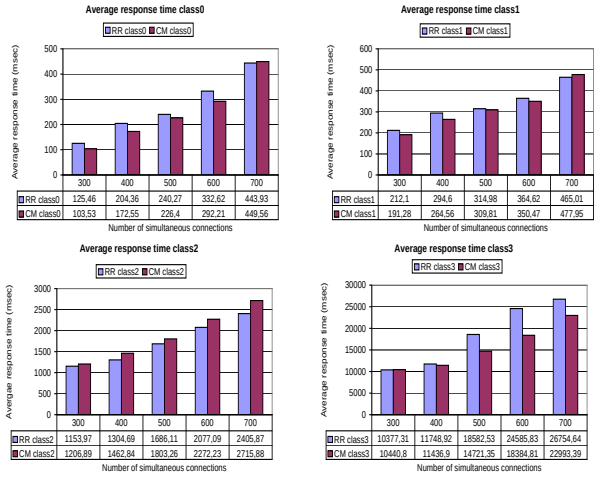

Figure 7. Average class response times for WebStone persistent HTTP requests

In general, the connection endpoint migration policy performs better for the classes 0, 1 and 3. The improvements of the classes 0 and 1 are significant for the light loads (300 and 400 simultaneous connections). The best gain for the class 0 amounts to 17.47% of the Round Robin time (for 300 simultaneous connections) while the best gain for the class 1 is 10.19% (for 400 simultaneous connections).

For all the loads, the class 3 figures for the connection endpoint migration policy are better than those of Round Robin. An interesting fact is that they are better as the load on the server increases. For 600 simultaneous connections, for instance, the class 3 servicing time 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 migration policy manages to save 3761.25 milliseconds, that is 14.05% of the Round Robin time.

7.5 Persistent HTTP connections evaluation of a three-node cluster-based server

We use the policy depicted in Figure 4 to migrate persistent HTTP requests for class 3 (large and unpopular documents accounting for 1% of the total number of the requests issued by the WebStone client). Like in the previous subsection, the class 3 has been experimentally identified to be the best-performing case. The policy operates on a three-node cluster-based server. The front-end redirects the persistent HTTP requests in a Round Robin manner to two of the server nodes. In turn, these servers migrate their connection endpoints to the third server for class 3 requests.

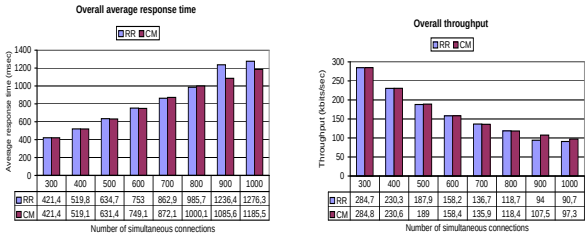

Figure 8. Overall performance figures for WebStone persistent HTTP requests in a three-node cluster-based server

The overall results are shown in Figure 8, while the detailed, per-class average response times are depicted in Figure 9. Essentially, Figure 8 tells that the connection endpoint migration policy that migrates persistent HTTP requests addressed to class 3 performs as well as Round Robin. For heavy loads, 900 and 1000 simultaneous connections, respectively, it even outperforms Round Robin.

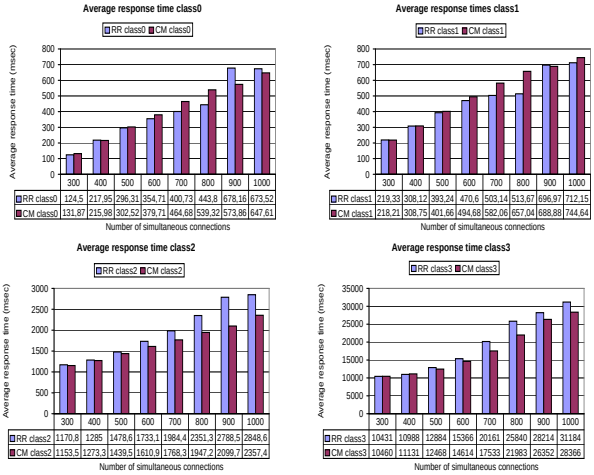

Figure 9. Average class response times for WebStone persistent HTTP requests in a three-node cluster-based server

Figure 9 explains this result. For all the loads but the 900 and the 1000 ones, the connection endpoint migration policy outperforms Round Robin for the classes 2 and 3, while exhibiting a poorer performance for the classes 0 and 1. For the last two loads, all the class results are better for the connection endpoint migration policy (except for the case of class 1 and 1000 connections, but even then the difference is minimal). Moreover, the class 2 and 3 (large and unpopular documents whose servicing time accounts for most of the total servicing time) figures show clear improvements over Round Robin. For instance, the connection endpoint migration policy operating on 900 simultaneous connections improves the class 2 average response time by 688.82 milliseconds (24.70% of the corresponding Round Robin time) and the class 3 time by 1861.99 milliseconds (6.59% of

the corresponding Round Robin time). For 1000 simultaneous connections, the class 2 average response time is with 491.23 milliseconds (17.24% of the corresponding Round Robin time) smaller than that of Round Robin, while, for class 3, the difference is of 2817.89 milliseconds (9.06% of the corresponding Round Robin time).

8 Summary and future work

In this paper, we presented a mechanism that migrates server-side TCP connection endpoints between two nodes in a cluster-based server. The protocol is client-transparent and supports back-end level request distribution policies. The server-side migrations run over high speed SANs whose performance makes the migration mechanism suitable for request distribution policies. For instance, simple locality-aware request distribution policies using connection endpoint migration show significant improvements over well load balanced policies like Round Robin. For non-persistent HTTP connections, one of our policies, operating on a three-node cluster-based server, improves the average response times for the small requests by factors of 2.79 and 2.25, respectively, while improving the individual request figures for the other document classes at the same time. In terms of the overall average response time, our policy outperforms Round Robin by a factor of 1.76. The TCP connection endpoint migration is also effective for persistent HTTP connections by improving the average response times for large and unpopular requests by as much as 25.22% of the corresponding Round Robin time.

In the future, we would like to further explore the scalability of the locality-aware request distribution policies using TCP connection endpoint migration. In particular, we would like to assess whether a two-level migration policy like that described in Figure 4 scales gracefully. For such policies, we would also like to find the optimal ratio between the number of the back-end servers that migrate some of their connection endpoints and the number of those that do not migrate requests.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

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

[17] 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.

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