SPECULATIVE TCP CONNECTION ADMISSION USING CONNECTION MIGRATION IN CLUSTER-BASED SERVERS

Vlad Olaru
University of Karlsruhe, Computer Science Department
Am Fasanengarten 5, D-76128 Karlsruhe, Germany
olaru@ipd.uka.de

Walter F. Tichy
University of Karlsruhe, Computer Science Department
Am Fasanengarten 5, D-76128 Karlsruhe, Germany
tichy@ipd.uka.de

ABSTRACT

This paper presents speculative TCP connection admission, a mechanism for improving sub-optimal request distribution decisions in cluster-based servers. Overloaded server nodes in the cluster speculatively accept incoming requests only to offload them to less-loaded nodes. Speculative admission uses connection endpoint migration, a technique that allows server-side connection endpoints to be arbitrarily assigned to server nodes in the cluster. Speculative connection admission targets distributed load balancing policies at the back-end level that leverage request routing decisions taken outside the cluster. The mechanism has been implemented in the Linux kernel as part of a policy-based software architecture for request distribution. We have been able to show that speculative connection admission adds little overhead to the normal TCP processing, offsets load imbalances and accommodates higher request rates. That makes it suitable for an active role in request distribution in cluster-based servers.

KEYWORDS

TCP networking, operating systems, cluster computing.

1. INTRODUCTION

Today, big “computer farms” are common place in the server industry. Such servers use multi-tier architectures, each tier being specialized for a given task. All these architectures employ at least two types of computers: front-end and back-end computers. The front-ends, also known as switches, redirect incoming requests to the servicing computers (back-ends). Most of the time, the front-ends drive the request routing according to load hints provided by the back-ends. Sometimes, front-ends use content-aware dispatching by inspecting information specific to the application-level protocol used. A lot of work has been done both in the area of load-balancing the back-ends and content-aware routing, but, with the advent of COTS clusters (clusters of Commodity Off-The-Shelf computers), there is evidence that improvements in the server operation may come from an enhanced cooperation among the back-end servers. Indeed, various developments in the area of distributed operating systems, including user-space communication subsystems (see von Eicken et al., Dubnicki et al., Myricom Inc.), software DSM systems (see Amza et al., Zhou et al.), cooperative caching systems (see Dahlin et al., Sarkar and Hartman), motivate the idea of moving request distribution at the back-end level. That would imply however moving load balancing mechanisms at the back-end level as well. One further incentive to this decision is represented by the fact that COTS clusters augment the traditional networking capabilities (Ethernet LAN) of a distributed server by those of a System Area Network (SAN). SANs have latency and bandwidth figures comparable to those of memory subsystems which advocates for a tighter node integration through cluster-wide Single System Image (SSI) services.

Cooperation at back-end level may involve changes in the server applications, but yielding good performance is only possible by changing the operating systems as well. Conventional standalone kernels fail to match the challenges raised by distributed computing and require new system level mechanisms and abstractions. One such SSI construct, helpful in back-end level request distribution, is connection migration, a flexible way of assigning server-side connection endpoints to particular back-end machines. Connection migration 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.

In this paper, we investigate speculative TCP connection admission, a back-end level load balancing mechanism either targeting fully distributed cluster servers (i.e., without front-end) or acting as a companion to front-end request dispatching. Speculative admission builds upon connection migration by speculatively accepting incoming requests on overloaded nodes only to further migrate them to less-loaded servers in the cluster. Sub-optimal routing decisions (either at the front-end or through some other mechanism in the case of fully-distributed servers) 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 for improving the situation. However, as soon as there are less-loaded nodes, speculative admission offloads the server by migrating connections to these nodes and thus offsets the imbalance. From this perspective, speculative admission acts as a load balancing mechanism or, for servers using front-ends, leverages load balancing at the front-end. Acting at the back-end level, speculative admission has the advantage of being fully distributed. Speculative admission is a mechanism and not a policy. Identifying lightly-loaded nodes in the cluster and developing methods of disseminating this information throughout the cluster are no topics for this paper.

2. SERVER ARCHITECTURE OVERVIEW

In this section we briefly summarize the features of our cluster-based server and describe the connection migration protocol. This is important in order to understand how speculative connection admission works.

From a hardware point of view, our cluster relies on the capabilities of a SAN acting as a communicating backplane among the cluster nodes. Most of the intra-cluster specific protocols are run over this communication backplane, while the interface to the “outside world” uses the LAN. The server may or may not employ front-end machines as *TCP-layer* switches. In the fully distributed version, each server node (i.e., a back-end) acts both as a connection router and as a server. A global server *Virtual IP* address is assigned to each of the back-end machines as an *IP alias*. A (*Virtual IP*, *service port*) pair defines a *generic* TCP endpoint. Connections linked to this generic endpoint can migrate from one physical server to another through the connection migration algorithm we will briefly present in the next subsection.

The request distribution system is independent of any particular service. It is based on classes of mechanisms like cooperative caching (see Olaru and Tichy) and connection endpoint migration, implemented as operating system pluggable services (modules). Our design aims at a deeper collaboration between the applications and the underlying kernel. Applications express their interest in request distribution through policies. Akin to extensible/grafting kernels (see Dahlin et al., Bershad et al.), these policies may be downloaded into the kernel at will. Therefore, our request distribution is *policy*- and not *service-oriented*.

2.1 Connection endpoint migration

Our connection endpoint migration protocol allows a flexible assignment of generic TCP endpoints to particular physical servers. This assignment is oblivious to the client; the entire server architecture is hidden behind the generic TCP server endpoint. Clients connect to this endpoint and potential migrations on the server side remain undetected from the client perspective.

Connection migration involves at most two steps. In the front-end version, uninformed (blind) connection routing chooses a given target endpoint for incoming requests. That assumes mapping the generic server endpoint to a physical one. For fully-distributed servers, this mapping step is implicitly performed by some other entity, say, a DNS server in Round Robin DNS (see RFC 1764). Once a connection is established, the server may decide to migrate its endpoint to another node in the cluster according to a certain policy. This is accomplished by fulfilling a migration protocol between two server nodes. For the front-end version, update packets are sent to the front-end in order to change the generic-to-physical endpoint mapping. For the fully-distributed case, the server initiating the migration keeps around a connection mapping table.

Technically, uninformed routing snoops incoming packets passing through the computer (either front- or back-end) and maps the Virtual IP of the server onto a Medium Access Control (MAC) address in order to deliver them to a given server. For front-end based servers, this is an Ethernet MAC address. Otherwise, it is an interconnect (in our case, Myrinet) MAC address. The mapping is either performed according to some hashing function (regardless whether load-aware or not) in the front-end case or by keeping around connection-to-MAC mappings, otherwise. Front-ends route incoming packets but not necessarily outgoing ones. Only 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.

The migration protocol runs over the SAN and relies on a checkpoint established at the server node currently handling the request. The checkpoint is built at connection setup time and stores incoming packets. It is periodically cleaned up as the request processing is carried out. When migrating a connection, the checkpoint content is flushed out to the new location of the migrated endpoint. There, it is replayed in order to mimic receiving the request at that node. In turn, the user-space daemon executes the application-level protocol. At the initiator node, a locally-prepared RST segment is used to “discard” the connection. In order to be able to replay the checkpoint at the migration target site, the migration initiator starts by carrying out the “three-way handshake” connection setup protocol (see RFC-TCP-81, Stevens) as if it would be a client requesting a new connection to the migration target node. For that purpose, the initiator of the migration uses properly set sequence numbers which, once accepted by the peer, will allow a safe replay of the checkpoint.

If the new server does not accept the migrated connection, a fall-back mechanism is used in order to resume the execution at the old server. The server triggering the migration is signaled the error and, normally, request processing is resumed from where it was left. Anyway, as soon as the server has been informed about the connection migration abort, the connection is again viewed as a regular, locally-bound connection.

3. SPECULATIVE CONNECTION ADMISSION

In cluster-based servers employing front-ends, 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 load of the back-ends. By inspecting the contents of the request, informed routing provides better amortized performance by taking into account additional information such as locality of the requested data. In both cases however, sub-optimal routing decisions may result in severe load imbalances among the back-ends. One legitimate question is, can we do better once such a decision caused a newly incoming request to be sent to an overloaded server?

Fully-distributed cluster servers do not employ front-ends to assign connections to back-end machines. They usually rely on an external entity to perform that mapping (round robin DNS, for instance) or perform the dispatching themselves. In the first case, external entities performing the routing may not be aware of the distributed server status and thus inflict severe load imbalances. How is it possible to offset such imbalances at back-end level? Servers performing the dispatching themselves are most of the time “virtual servers”, i.e., one-IP distributed servers relying on MAC-level routing. However, this aspect makes them hardly amenable to content-aware routing. How is it then possible to correct routing decisions at later stages?

Speculative connection admission tries to answer these questions. The idea is to additionally accept 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 connection migration.

From a technical perspective, this is accomplished the following way. When using the standard socket library, a server declares its interest in servicing requests in two steps. First, it invokes a *listen* system call that establishes a queue in the kernel for incoming connection requests. Then, through an *accept* system call, the server picks up established connections from that queue and passes on the corresponding connection handle to the part of the server program servicing the requests. Speculative connection admission is made possible by enlarging the queue storing incoming connection requests. The “extra-accepted” connections are marked migratory. A request distribution policy will migrate the connection according to either cluster-wide load hints or the locality of the referenced data.

In general, connection migration can be regarded as the driving engine of distributed request dispatching at the back-end level. Postponing routing decisions until requests hit the back-ends is an important issue for solutions striving 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: CPU utilization of the nodes in the cluster (for load balancing policies) or globally cached data (for content-aware routing). In this regard, speculative connection admission further extends the capabilities offered by connection migration and acts as a support for back-end level policies leveraging routing performed outside the cluster. However, this paper does not discuss methods to identify lightly-loaded servers (depending on the metric used, a lightly-loaded node can be one with small CPU utilization, small number of established connections, small I/O activity, or combinations of all these). In the next subsections, we describe the implementation of speculative connection admission in the kernel and how to use it together with the policy concept of our request distribution system.

3.1. Operating system internals

Modern operating system kernels use an event-driven model to process network traffic. Network card interrupt handlers store packets in a general purpose queue and schedule 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 later processed by the targeted process when running in kernel mode as a consequence of executing a system call (for our example, either *accept* or *read/readv/recv/recvfrom/recvmsg*). This event driven model aims at minimizing the protocol processing overhead, as the targeted process doesn't need to actively wait incoming packets (polling is a bad idea for non-preemptive kernels like Linux, at least the versions before 2.6).

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 aforementioned event-driven model. In particular, it responds immediately with the appropriate SYN_ACK and adds an open-request structure to a *listen* queue associated with the listening socket. As soon as the client sends back the ACK to the SYN_ACK, the corresponding open-request 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 is released.

Although a single queue (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 actually considers only the “accept” part of the queue.

3.2. Speculative connection admission policies

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 about an additional “admission” queue. Practically, this 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 final ACK in the “three-way handshake” connection setup protocol arrives at the server side, 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 (that is, including the “admission” queue).

Once the established connection has been accepted by the server software (as a result of an *accept* system call), the packets of a request are gathered in the receive queue of the socket. This queue is part of the checkpoint that migratory connections administrate (see Section 2.1). As soon as the request (protocol command) is built in memory, the server application can access the data through *read/readv* or *recv/recvfrom/recvmsg* system calls. When running in kernel mode, these system calls invoke *tcp_recvmsg*, a routine that checks the receive queue and transfers the data to user-level. The speculative policy operates inside *tcp_recvmsg* by avoiding the data transfer to the local server in order to migrate the connection. Namely, the speculative policy migrates the socket previously marked migratory by calling the migration routine and closes the local connection endpoint (by simulating the reception of an RST segment).

One legitimate question is why should the speculative policy wait for the server application to try to process the request before migrating the connection. The answer is that meaningful decisions might be possible only by inspecting the request. For instance, a content-aware policy might find out the file to be accessed and, depending on the actual location of cached copies of the file, a performant decision may be taken. However, all this comes at a price. Modern server programs use multiple threads (sometimes even processes, as in Linux) to service 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 has been spawned and scheduled for execution. Even though many server programs use pre-spawned threads in order to avoid the overhead of thread creation, there remains to pay the price of an additional context switch.

One other important issue regards the migration rate. Assuming that servicing a request takes more time that speculatively admitting and migrating it, it means that the node that speculatively admits 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., shrinks the admission queue to zero) and waits for a notification from the migration target before it starts again to speculatively accept connections. 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 speculative admission has a null window, then it 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 approximates the number of migrated connections that the migration target might have processed in the meantime).

4. PERFORMANCE EVALUATION

We used the S-Clients benchmark (see Banga and Druschel) to evaluate the performance of speculative connection admission. S-Clients is capable to generate HTTP request behaviors in bursts as typically seen in the Internet but tests only the speed of the server software (including the kernel TCP/IP stack) without paying attention to additional issues (caching, for instance). The benchmark generates requests targeting a single file, *index.html* (roughly 4KB in size). All the experiments used a front-end based server. The fully distributed server case can be derived from this one by considering the front-end to be the external entity dispatching requests.

Our experiments consist in launching two S-Clients processes on a client machine (called C). The requests are redirected by the front-end to a server machine called A. At this point, we explored two scenarios. In the first one, machine A serves the requests itself. This serves two purposes: as a baseline case for the comparisons with the next scenario and also as a way of understanding the impact of varying the accept queue size on the server software. In a second scenario, server A uses speculative connection admission to offload some of its connections onto the other server (B). This second scenario intends to evaluate the operation of speculative admission in real-life conditions, as server A handles a given number of connections and migrates the speculatively accepted ones to server B. To assess the load imbalance impact on the server performance, we instructed the front-end to route requests asymmetrically: two thirds to server A and one third to server B and three quarters to server A and one fourth to server B, respectively.

4.1 Experimental Setup

The back-end servers are 350 MHz Pentium II PCs with 256 MB of RAM, 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 and 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 (see Myricom Inc.). Each back-end runs Apache 1.3.14 (see Apache) as 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 run Linux 2.4.19. All the machines are interconnected through regular 100Mb/s Ethernet (with the front-end acting as router between the client and the servers).

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

We started our experimental evaluation by testing the performance of a standalone server in terms of achieved connection throughput when varying the size of the "accept" part of the listen queue. The results are presented in Figure 1. The SYN_RCVD queue length 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 the Figure 1, notice that the best performing case is "single.128-96", while the largest performance degradation is for an accept queue length of 64. Since we want to test the performance of speculative connection admission under heavy load conditions, we chose "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.

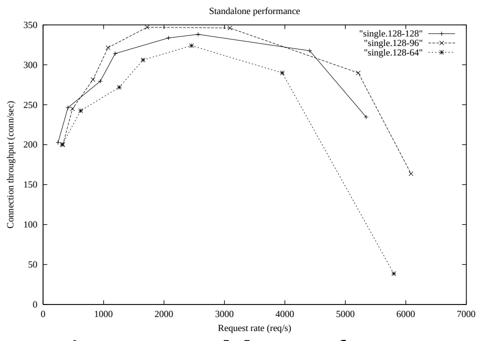

Figure 1. Standalone performance

4.3 Speculative connection admission in request distribution

Both connection migration and speculative connection admission are meant to be used by request dispatching algorithms. On our second experiment a server speculatively accepts connections only to migrate them to another server. We report connection throughput values for normal processing (i.e., without speculative admission) and speculative admission. 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"). 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 (e.g., "cm33" means that speculative admission using connection migration operates on a 64 + 32 long accept queue, in which the admission queue represents 33% of the total queue).

The first experiment (in which two thirds of the requests hit server A and one third server B) uses an admission queue of 32 entries for our speculative admission policy. The results are reported in Figure 2 (the curves labeled "w33"). In Figure 3 we present the results for the experiment instructing the front-end to route three quarters of the requests to server A and one quarter to server B (curves labeled "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.

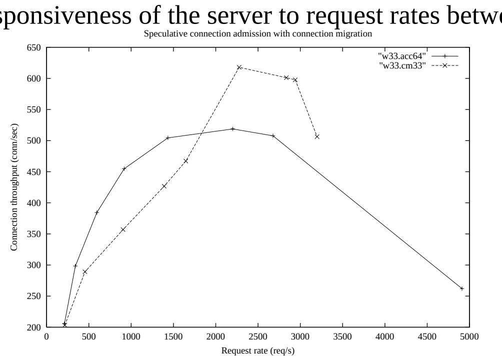

Figure 2. Speculative admission in request distribution (front-end routes 33% of the requests to one server and 67% to the other)

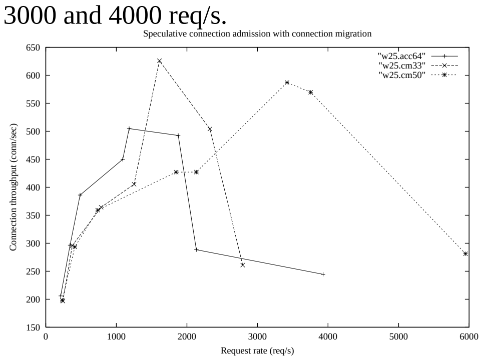

Figure 3. Speculative admission in request distribution (front-end routes 25% of the request to one server and 75% to the other)

For the case in which the front-end routes 67% of the requests to one server and the rest to the second server (Figure 3), it is worth noting that for smaller 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 increased demands. It yields the peak performance at about 617 conn/s and some 2200 req/s. Nevertheless, 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 3). Indeed, both speculative cases outperform normal processing (denoted by “w25.acc64”) for high request rates. Also notice that, for high request rates, larger admission queues yield significantly better connection throughput figures. Similar to the first experiment (Figure 2), at small request rates normal processing handles better the situation, as the overhead of connection migration cannot be hidden.

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 1, “w33.acc64” in Figure 2 and “w25.acc64” in Figure 3, one can see that the peak performance of a single server is at about 324 conn/s while none of the two cases for the two-server achieves more than 519 conn/s. It is the job of speculative admission to overcome the imbalance and to improve the performance by almost doubling the peak connection throughput figure.

5. RELATED WORK

Connection migration is a fairly new mechanism put under scrutiny by Snoeren et al. and Sultan et al. The first solution is not even a true migration protocol as it involves an user-level “wedge” intermediating between the connection endpoints. Moreover, their protocol is application-dependent (i.e., not a TCP-migration protocol). Both papers 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. This solution does not need front-ends or switches between the client and the server. However, it is unclear whether it can successfully be 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-grained fail-over mechanism for long-running connections switching across a distributed collection of replica servers.

Our server-side (server-to-server, client-transparent) connection endpoint protocol is application-independent. It has both versions for architectures employing front-ends and for those fully-distributed. Since it is oblivious to the client, the protocol has the advantage of not being sensitive to the Internet behavior while benefiting of powerful interconnects which have better performance figures than the Internet. The disadvantage is the use of an additional connection router (either the front-end or the server node itself). For servers acting as connection router the issue is mitigated by the low message passing cost of SANs.

According to a taxonomy proposed by Cardellini et al., our server architecture is a mixture between *cluster-based servers* and *Virtual servers*. Unlike cluster-based servers, our server does not assign the *Virtual IP* address to the front-end but rather to each back-end node, quite like in a *Virtual server*. However, our design may allow front-ends to participate in request dispatching, mostly due to their involvement in connection migration. Our request dispatching mechanism is hybrid, a combination of TCP-layer routing (either at the front-end or at the server node itself) with distributed policy-oriented routing at the back-end level by using connection migration.

6. CONCLUSIONS

In this paper we presented a mechanism for speculatively admitting incoming TCP connections for overloaded nodes in a cluster-based server. Such nodes may decide to use a connection endpoint migration protocol to offload some of their requests to other cluster nodes. The main goal of our work was to assess whether this mechanism can be used actively in request distribution in cluster servers. The results of our tests using a small test-bed cluster encourage us to positively answer the previous question. We described also the general architecture of the request distribution algorithm of our cluster-based server and pointed out how speculative connection admission may be useful in such a context.

REFERENCES

Amza, C. et al, 1996. TreadMarks: Shared Memory Computing on Networks of Workstations. In IEEE Computer, Vol. 29, No. 2, pp. 18-28.

Apache. http:/www.apache.org/

Banga, G. and Druschel, P., 1997. Measuring the Capacity of a Web Server. In Proceedings of the Usenix Symposium on Internet Technologies and Systems.

Bershad, B. et al, 1995. Extensibility, Safety and Performance in the SPIN Operating System. In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15).

Brisco, T., 1995. DNS Support for Load Balancing. RFC 1764.

Cardellini, V. et al, 2002. The State of the Art in Locally Distributed Web Server Systems. In ACM Computer Surveys, Vol. 34, No. 2, pp. 263-311.

Dahlin, M. et al, 1994. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In The First Symposium on Operating Systems Design and Implementation.

Dubnicki, C. et al, 1997. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In Proceedings of Hot Interconnects V.

Engler, D. et al, 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In Proceedings of the 15th Symposium on Operating System Principles (SOSP '95), Copper Mountain Resort, Colorado, USA, pp.251-266.

Postel, J. Editor, 1981. Transmission Control Protocol. RFC 793.

Olaru, V. and Tichy, W., 2003. CARDs: Cluster-Aware Remote Disks. In Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2003).

Sarkar, P. and Hartman, J, 1996. Efficient Cooperative Caching using Hints. In Proceedings of the Second Symposium on Operating Systems Design and Implementation.

Snoeren, A. C. et al, 2001. Fine-Grained Failover Using Connection Migration. In Proceedings of the Third Annual USENIX Symposium on Internet Technologies and Systems (USITS).

Stevens, R. W., 1994. TCP/IP Illustrated, Volume 1. Addison Wesley Longman, Inc.

Sultan, F. et al, 2001. Migratory TCP: Highly Available Internet Services Using Connection Migration. Technical Report DCS-TR-462, Department of Computer Science, Rutgers University, Piscataway, NJ 08854-8019.

Myricom Inc. GM: The Low-Level Message Passing System for Myrinet Networks. http:/www.myri.com/scs/index.html.

Eicken von, T. et al, 1995. U-net: A User-Level Network Interface for Parallel and Distributed Computing. In Proceedings of the 15th ACM Symposium on Operating System Principles.

Zhou, Y. et al, 1996. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In Proceedings of the 2nd Symposium in Operating Systems Design and Implementation.