A Scalable Approach for the Distribution of E-commerce Services Based on Application Level Active Networks

Fabrício Benevenuto, Breno Vitorino, Bruno Coutinho, Dorgival Guedes, Wagner Meira Jr.

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627, Belo Horizonte, MG
CEP : 31270-901

{fabricio,vitorino,coutinho,dorgival,meira}@dcc.ufmg.br

Abstract. *In this paper we use the idea of Application Level Active Networks to implement an efficient and easily deployable solution for the distribution of e-commerce services using cache servers which can hold dynamic content. The resulting system is an application of the ALAN concept and framework to a real and complex application (an electronic bookstore that makes use of dynamic document caching schemes). Our experiments show that significant performance gains can be obtained by e-commerce sites that use this model, improving the scalability of the target service by distributing the load among the dynamic caches. Results show reductions of server CPU load by up to 64% and improvements of client perceived response time of up to 70% under heavy load.*

1. Introduction

Recently, the World Wide Web (WWW) has become an important way of providing services and transactions according to different negotiation models. In particular, electronic commerce (e-commerce) is responsible for a major fraction of this growth, becoming one of the driving forces behind the expansion of the Internet and the number of users and companies that use it on a daily basis.

In general, the growth of electronic commerce has led to high demands on sites that provide such services. As a consequence, servers often get overloaded, increasing significantly the waiting time of end users, despite the efforts to improve the efficiency of servers. To meet the quality of service demanded by a growing number of on-line customers, e-commerce services must be implemented with scalability in mind.

A common and successful strategy for improving the scalability of WWW sites is the use of caching in the network by the deployment of Web proxies. Such servers can be seen as inter-mediator for all traffic between clients and HTTP servers. Proxy servers are often used to bound traffic from a portion of network, inside which nodes can be treated as being topologically (and many times geographically) close to each other [Abrahao and Benevenuto, 2003]. In such an organization, every request started by a client and every response from the Web servers pass through the proxy server. When

a client requests a file, not only does it forward the request to the destination Web server, but it also handles the incoming response in the opposite direction. In addition to forwarding the received file to the client, it stores a copy of it in its local cache. That copy enables it to fulfill future client requests for the same file without the need to contact the end-server again.

Based on that principle, proxy servers would look as a reasonable option to offload some of the processing required by an electronic business server, working as an outpost closer to the clients. That would in turn lead to a more scalable architecture, since it would distribute processing among the server and various proxies. Since proxy servers are usually installed in Internet Service Providers (ISPs) and other heavy-traffic network locations, they would be well suited to handle part of the requests and offload not only the server, but the network connections behind them. Furthermore, proxy servers can usually take advantage of similarities among customers of a given community. For instance, a proxy server may concentrate all traffic generated by employees of a given corporation, who may share some interests, increasing the number of requests served by the proxy alone.

However, although caching is a proven technology for static objects (e.g. HTML pages and images), its use for dynamic objects, which are very common in e-commerce servers, is yet to be solidified, despite some important research efforts in the area [Cao et al., 1998, Meira Jr. et al., 1999].

The difficulties to deploy caches for dynamic objects have various reasons. One important concern that must be considered is *deployability*, since handling dynamic objects require special solutions besides the usual facilities provided by caching architectures. Many of the solutions already proposed require changes to the client software, or to the network infrastructure, or the servers, or to all of them. The large number of deployed clients and networks in the Internet makes it difficult to change clients, and even more difficult to promote changes to the networking infrastructure (as we can see by the effort to deploy IPv6). On the other hand, the increasing complexity of servers (even without considering such facilities) makes it hard to deploy solutions that require server changes [Gribble et al., 2000]. What we need, therefore, is a technique that does not require changes to the existing infrastructure (client browsers, network routers and switches, servers, and protocols).

In this paper we propose a technique to improve the quality of service of e-commerce services and make them more scalable by the use of caching. The idea behind the technique is to keep in proxies the means to perform non-critical tasks such as searching, which would leave e-commerce servers to focus on critical tasks such as registering selections and orders and handling payments. To that end we make use of an application level active network (ALAN) framework which provides us with a deployable and portable environment to distribute the e-commerce operations.

The ALAN concept has been developed exactly to allow an easier deployment and expansion of new technologies in face of the growing ossification of the communication infrastructure [Fry and Ghosh, 1999]. In this respect, it is only natural that an ALAN framework be used in proxy servers as an environment to implement distributed

e-commerce applications extensions such as dynamic caches. Such applications can provide better performance by making use of frameworks that exploit the main features of ALANs, contributing to improve the scalability of e-commerce servers.

Various studies [Fry and Ghosh, 1999, Liabotis et al., 2001] have presented the application level active network architecture as a means to provide flexibility to the communication infrastructure. Definitely, ALAN is a promising architecture that allows easy dissemination of new functionalities, supplying deployability and safety to mobile code. However, to the best of our knowledge, this architecture has never been applied to large applications, particularly to commercial systems. That is done in this work, where we use ALANs as an environment to distribute e-commerce services to proxy servers. We developed a dynamic cache system on top of that architecture. We compare the performance of the developed system with existing known proxy cache architectures to show that the ALAN environment can be used efficiently to reduce user-perceived latency, network traffic and server load.

The rest of the paper is organized as follows. The next Section presents related work on mechanisms for distributing dynamic Web server processing and the related load in order to improve quality of service. Section 3 presents the architecture and major components of an e-commerce service. After that, the ALAN architecture is described in Section 4. Section 5 discusses the advantages of using the ALAN framework as an environment to distribute e-commerce servers. In Section 6 we present the experimental methodology used in the experiments and in Section 7 we evaluate the performance gains of our approach.

2. Related Work

The idea of distributing the computation associated with dynamic documents is not new in itself, and a few different solutions have been discussed in the literature. One approach is to migrate some services to the clients through applets [Vingralek et al., 1999, Yoshikawa et al., 1997], improving the quality of the services provided. This approach is applicable to various Internet systems, where computation can be delegated to the clients. However, it is not applicable to e-commerce services, since the reference locality of accesses of a single user does not justify the cost to download complex applets. Other works have demonstrated the benefits of adding transparent services to clients and servers [Silva, 1998]. Our framework also permits transparency and dynamic deployment of services, and make that in a portable way.

Another approach is to distribute e-commerce services by making use of proxy servers, reducing customer latency and bandwidth consumption at e-commerce sites [Cao et al., 1998, Meira Jr. et al., 2000]. Results presented in the literature show the potential benefits of such solutions, which are shown to reduce server load and improve the quality of service. However, the solutions presented by Cao *et al.* and Meira *et al.* both depend on changes to the proxy architecture and/or servers, which make them hard to deploy in the general case. Our work uses the same idea of distributing dynamic document processing, but by the use of a deployable environment, based on an Application Level Active Network architecture.

The traditional active network proposal [Fisher, 2001], relies on dynamic programmable routers or switches to host and execute mobile code. Various studies [Fry and Ghosh, 1999, Liabotis et al., 2001] have shown the advantages of the use of this architecture as a means to provide efficient abstractions, deployability and security to mobile code. However, real active network applications have not emerged yet, in part due to the lack of an appropriate and widespread infrastructure. Application level active networks have emerged as an attempt to grant more security and flexibility to the original active networks. However, the viability of real applications has not been observed since the concepts were presented through the use of very simple applications [Fry and Ghosh, 1999, Liabotis et al., 2001, Amir et al., 1998]. In this sense we believe our work presents a significant contribution by showing the use of ALAN as a platform in which complex and useful applications, such as distributed e-commerce services, can be executed.

3. E-Commerce Services

In order to understand how to distribute e-commerce services we must understand the relations among the various entities involved and the architecture of the server as a whole. The following sections address those two issues.

3.1. E-Commerce Entities

There are basically three main entities to be considered in e-commerce: products, clients and sessions.

Products represent the commercialized elements of the e-commerce site. They can be classified basically in two kinds of data: static and dynamic. Static data comprises information that is not affected by the services provided by the e-commerce server, such as a book description. On the other hand, dynamic data contains all pieces of information that are affected by the operations performed while providing services, such as the number of books in stock.

Clients represent the customers that use the e-commerce site. Again, the server records static and dynamic data. In this case, information such as name and address are static, while other elements, such as the content of the shopping cart, are dynamic.

Sessions represent the interaction between clients and products. That interaction is usually represented as the user session, built by the server in response to user requests. Each session combines references to static and dynamic data of a client and some number of products.

3.2. E-Commerce architecture

Considering the operations performed by an e-commerce service and the different types of information that must be handled, the structure of a server is usually divided in three major components: the WWW server, the transaction server, and the database management system (DBMS) [Coutinho et al., 2002]. Figure 1 shows how these elements fit together.

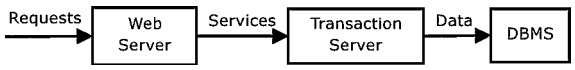

Figure 1: E-commerce service architecture

The *WWW server* is responsible for presenting the service, receiving user requests and parsing them. After that, the WWW server must manage all operations. Requests for static data are served directly, whereas requests involving dynamic data are forwarded to the transaction server. It is also the responsibility of the WWW server to deliver the page composed by the transaction server as a result of a dynamic request.

The *transaction server* (also known as the application server, or virtual store) is the heart of the e-commerce service. It is the component that must implement the logic of the business: how to handle client searches for products, how to present the products, how to complete a sale. It must keep track of the user sessions, so it can control the shopping cart, and provide site personalization, for example. Once it receives a command for a certain service from the WWW server, it must implement the logic of the operation. That may include accessing the database server to complete the request and build the response page, which will be handled back to the WWW server.

Most of the information related to the store operation must be kept consistent across sessions and in stable storage to guarantee proper operation over time. That is the responsibility of the *database management system*, the third element of the server. It holds product and client information, number of items available and other data. It receives commands from the transaction server and returns data as a result.

4. ALAN Architecture

This section describes the architecture of an Application Level Active Networking environment. Our architecture is very similar to the first proposed ALAN architecture [Fry and Ghosh, 1999]. However, our framework does not implement some features such as protocol servers and protocol stacks, since they are not needed in this case.

An ALAN environment can be understood as a user-level framework capable of downloading a specific program, developed by trusted third parities, and executing this application in a controlled environment that can grant security. Therefore, ALANs provide deployability and maintenance of services, without inflicting the existing infrastructure, since it works at the application level. In the present work, we used that framework as a proxy server to be able to execute distributed e-commerce services.

The ALAN system can be divided into a number of key components and concepts which are described below. Figure 2 shows how the components fit together. The proxylets constitute the application to be downloaded and executed, the actual mobile code. Since Java is the implementation language adopted, they are java byte-code, specifically developed to be executed in the Dynamic Proxy Servers (DPS) by including the appropriate DPS/proxylet interfaces. The Dynamic Proxy Servers must be strategically located over the network, in a way similar to the positioning of static proxy servers. The proxylets are stored in special servers, the Proxylet servers, from which they can be obtained.

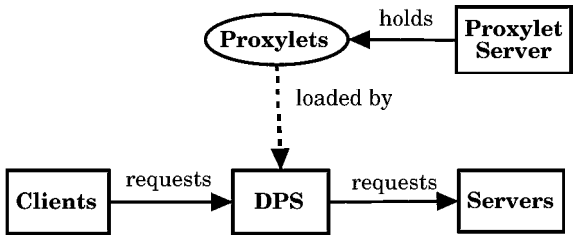

Figure 2: Architectural overview of an ALAN

The proxylets are downloaded to the Dynamic Proxy Servers on demand and can be used to intermediate communication between client and server. While active, they can parse incoming HTTP requests, checking if the user is accessing a service which the proxylet supports. Using this approach, various proxylets can be executed at the same time. A proxylet is totally self contained, with all the code required to perform its networking and filtering functions. In our application proxylets are simple jar files, a set of classes specifically developed to be executed onto the DPS.

Dynamic Proxy Servers (DPSs) accept proxylets which are executed on behalf of third parties. Communication with a DPS to control it is through Remote Procedure Call (RPC). Since Java is the current implementation language, Remote Method Invocation (RMI) is used [Sun Microsystems, 1998]. The Dynamic Proxy Server accepts a simple set of requests through a RMI interface to control the use of proxylets, which can be accessed by clients as needed. The requests are as follows:

Load: A reference to a proxylet is passed to the server. A number of validation checks are made at this stage. The server may be heavily loaded, so it may not be in the position to accept any more requests. It may be the case that proxylets can be downloaded only from prescribed servers. A service provider may only allow a controlled set of proxylets to be executed onto its server. If the proxylet is acceptable after all validation checks it is loaded into the server. As the proxylet is passed by reference, it may be the case that the proxylet has been loaded previously and may be cached on server already, so no further downloading is necessary.

Run: Once the proxylet has been loaded it is necessary to start it.

Modify: It may be necessary to change the state of a running proxylet. For instance, one may need to change the proxylet parameters during its execution.

Stop: In some situations it may be necessary to force the termination of a running proxylet.

The Proxylet server is a repository of proxylets. The DPS administrator is able to download proxylets from it by just passing an URL to the DPS. Loading proxylets from common Proxylet servers presents two main advantages. First, the proxylet code can be shared by other DPSs since it is loaded from a known URL. Second, it can be used to guarantee that proxylets are sourced from trusted servers. It is the job of proxylet servers to verify that nodes requesting a proxylet have the right access rights to download it, as well as to provide the DPS with signature information about the available proxylets, for example.

This architecture enables rapid deployment of new communication services on demand and is portable across heterogeneous hardware and software platforms. Moreover, Dynamic Proxy Servers and the associated proxylet servers can, typically, be owned by a network provider or the owner of a private Intranet, which simplifies deployment and administration even further.

5. Using ALANs to Improve E-commerce Servers

The DPS can be strategically deployed on an existing proxy server. In this way, some specific requests of the clients that the traditional caches cannot answer can be handled to the proxylets running on the DPS. The proxylet application depends on the kind of data it will handle and the application logic. In this context, the identification of static and dynamic elements is crucial for the distribution of the e-commerce service. While static data can be easily replicated among different DPS, each dynamic piece of data must be properly identified and any access to it must be properly controlled to avoid inconsistencies. In particular, there are four properties that are essential to distribute e-commerce operations using an ALAN environment:

Transparency: Most WWW browsers have the ability to be configured such that all transactions are performed through a local cache. Rather than have all HTTP requests going directly to the server which is pointed to a requested URL, they first go to a local cache. The setting in the WWW browser to point to a cache is typically called the proxy cache. In this way, the architecture presented in section 4 allows complete transparency among clients and servers. The proxylets, executing on a DPS, analyze the client requests when they pass through the DPS within the proxy server and are able to take some pre-programmed decisions based on the requests contents. One of these decisions might be to get the result of a search from the server. In this case, the server treats the DPS as a common client.

Deployability: Our framework interfaces with proxy servers through existing mechanisms. Both clients and servers do not suffer modifications. Moreover, our framework is portable and can be easily deployed on a proxy server.

Safety: Although DPS servers provide security against several proxylet operations, the proxylets can be stored in common servers in order to guarantee that the proxylets are sourced from trusted servers. Furthermore, common servers allow proxylet code to be shared.

Consistency: Information provided by the dynamic cache on the proxy server should be consistent with the server state. There are studies that propose different policies to maintain data consistency [Meira Jr. et al., 2000]. This consistency depends on the kind of data the proxylet will use. For instance, remove and alter the description of a book are not frequent operations whereas the number of books in stock are constantly changed.

Based on these four properties we can see that this framework simplifies the application developed for this environment. The ALAN framework provides transparency and deployability, whereas the DPS and the proxylet servers provide safety. For instance, a proxylet that handles dynamic data must just worry about data consistency whereas transparency, deployability and safety are provided by the ALAN environment.

6. Experimental Methodology

This Section discusses the methodology used for the experiments we have performed. We have developed in our labs an e-commerce server which implement all the operations of an electronic bookstore. By using proxylets and DPS we augmented the service to include the use of dynamic caches. To understand how the whole experimental system works we first need to understand how the basic e-commerce server we implemented works. This server has an internal cache to store results of search operations. When a client searches for some word, the server first looks for the response in its own cache. If a miss happens, it requests the response from the database system. Instead of implementing a complex replacement policy, this internal cache system consists of a simple hash table limited by the available system memory. This e-commerce server is divided in two components: the Web server, integrated with the transaction server, and a database system (MySQL 3.23.51).

The developed proxylet is a dynamic cache system able to store results of searches submitted to the server. The results cached are book descriptions, considered static data. However, although that information can be classified as static, problems of inconsistence may occur in some situations. For instance, suppose that a new book is registered on the database management system. If the dynamic cache is not immediately informed about the new book, a search operation can result in an outdated list of available books. Operations of inserting and updating the database are not frequent; consistency could be granted by the use a dynamic cache programmed to empty its contents regularly. In our work we do not consider any algorithm to verify the consistency of search lists with the e-commerce server.

To evaluate our application, we have constructed a system composed by the e-commerce server and two proxy servers. We compare the use of our DPSs running the dynamic cache proxylet onto the proxy servers with the *Squid* cache [Wessels and Claffy, 1997] by itself. The gain of our approach is that the dynamic cache developed is able to cache data considered uncacheable by the Squid system. For instance, Squid considers results of *cgi-bin* applications and search operations of e-commerce servers as uncacheable. Our application keeps in cache results of searches, improving response time to the end-user and the scalability of the e-commerce server. Furthermore, the dynamic cache proxylet contacts the server, in case of a miss, through RMI. In this way, the e-commerce server does not need to execute some operations such to parse the request because the proxylet gets the necessary information to the server directly through RMI. This simplifies the task of the server, allowing it to handle other requests.

To simulate the clients we use two machines running httperf [Mosberger and Jin, 1998] directed to the Proxy Servers. Each httperf client is directed to one Proxy Server and each Proxy Server is running the DPS or the Squid Server. Figures 3 and 4 depict the scenario just described. The workload used on the httperfs is composed of several search operations from logs of a real e-commerce server. These operations represent a 36% of the original logs, which are characterized and thoroughly described in [Menascé et al., 2000]. Before running the experiments, we warm the caches with another workload, not used in the experiments.

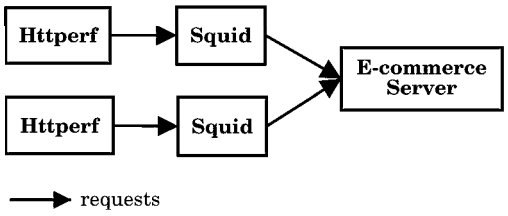

Figure 3: Experimental architecture using Squid as proxy

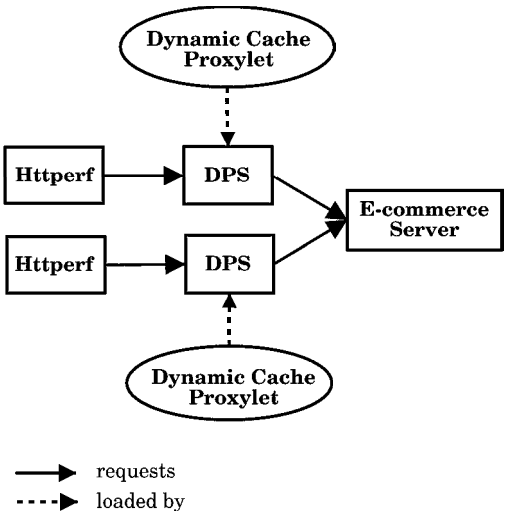

Figure 4: Experimental architecture using DPS as proxy

Although we have not evaluated the DPS and Squid working together, this architecture is perfectly possible. E-commerce server architectures are often divided in one machine for static requests and another for dynamic request. Squid would relieve the static server whereas the DPS would relieve the dynamic server. However, it is necessary to introduce a pattern matching engine to decide if the objects are static or dynamic, based on the requested URL. The overhead of this was not measured. We have concentrated our experiments to evaluate the impacts of using the DPS-based architecture on the dynamic server scalability.

The dynamic cache proxylet does not implement a replacement policy like Squid. Instead, this application implements a solution similar to the developed e-commerce server, with a hash table to store the results of the search operations.

The experiment results discussed in section 7 were obtained with five machines equipped with *Intel Pentium III* 750 MHz processors with 192MB of RAM executing the server and the proxies. The database server used a computer with a *Intel Pentium IV* 1.6 GHz processor and 1 GB of RAM memory. Each machine uses Linux 2.4.x as their operating system. All machines were connected by a *Fast Ethernet* switch at 100 Mbps.

7. Results

In this section we evaluate the benefits of using a dynamic cache proxylet running on the proxy server.

In order to visualize the efficiency of the DPS, we first compare the number of requests that arrive on the server when the proxy servers run just Squid or the DPS. Figure 5 shows the results. It is clear that the DPS-based system has a higher hit rate than the Squid-only setup, since the number of requests reaching the server is lower for the DPS, although both systems receive the same client load. Both curves tend to level off after ten

sessions per second, basically because the proxies saturate at that point. The curve for the proxylet-based solution shows a small drop at higher loads due to the implementation of the search list repository. After a certain point, the higher the request rate, the slower is the operation of the hash table holding the lists, due to the number of collisions, saturating the DPS server and causing a decrease on its efficiency. A cache replacement policy on the proxylet could have delayed this saturation.

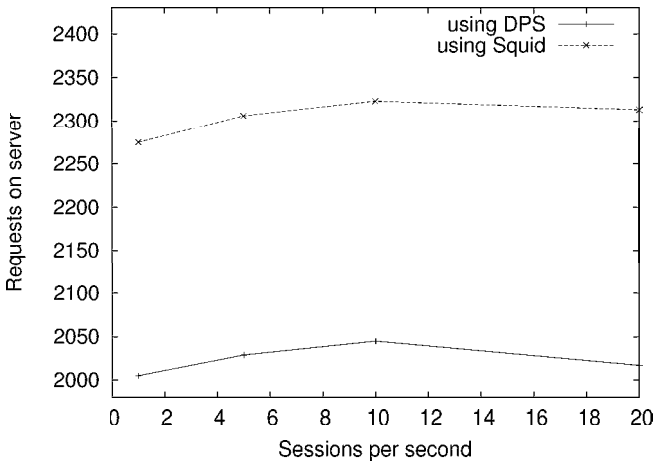

Figure 5: Number of requests that reach the e-commerce server (cache misses)

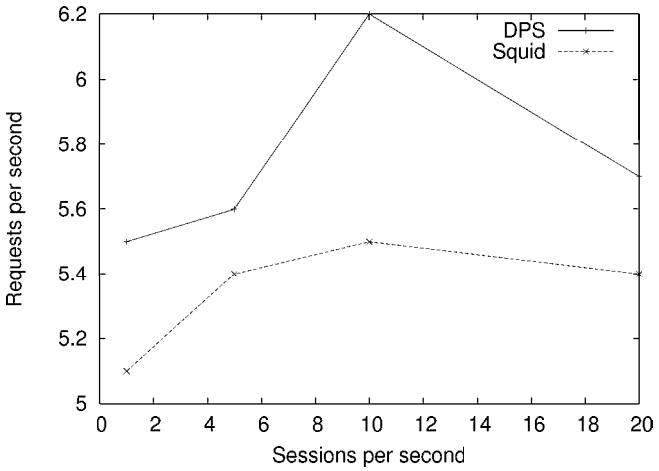

Figure 6: Requests per second achieved by clients. It compares the use of DPS and Squid as proxies

Figure 6 shows the request rate of the e-commerce server comparing the use of Squid and DPS running on the proxy servers. The Squid server does not cache any search results, so we can say that the Squid numbers reflects the server performance, which must answer most requests. Up to ten sessions per second, the request rate is increasing both with DPS and Squid, but the DPS always shows better, since they distribute part of the server load among the proxy servers, achieving better load distribution and scalability. With more than ten sessions per second, the Squid measures suggest that the server has

saturated. As with the proxylets, the e-commerce server stores search lists as a hash table. The same saturation observed on the e-commerce server occurs on the proxy servers, causing a decrease of the request rate. This effect can be better verified with the DPSs. These servers are contacted directly by the clients, and absorb a large part of the client requests using their internal hash table as cache. In this context, the DPS caches relieve the e-commerce server cache and delay its saturation. However, the proxy server caches saturate first themselves. The down slope on figure 6 is steeper on the server using DPS as proxy because of the overhead of the proxylet's hash table, which start to get swapped to disk by the virtual memory system.

Figure 7 shows the average response time of the requests. After ten sessions per second both, the server response time using DPS and Squid as proxy, remains almost constant, but times are much better (lower) for the DPS solution. This suggests that the server request queue is full, and the httperf cannot post more requests until the server is able to complete some outstanding operations. Otherwise, the server drops any extra requests it gets. For all plots, the DPS shows a smaller response time. In case of a cache hit on the DPS, it responds directly to the clients, contacting the server just in case of misses. On the other hand, Squid cannot cache search results, contacting the server more frequently than the DPS.

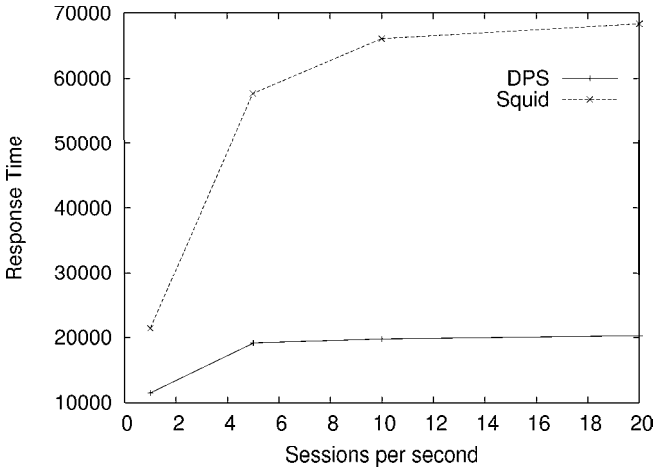

Figure 7: Response Time measured on clients. It compares the use of DPS and Squid as proxies

Another benefit of the use of proxylets is reducing the CPU load on the server, as can be seen in Figure 8. When the proxy servers are using Squid, the CPU utilization on the server is up to 2.5 times higher than with the proxy servers using DPS. This reflects the fact that Squid cannot cache search results, leaving all those operations to the server. The DPS can distribute the load of the search operations between the proxy servers, relieving the e-commerce server. The dynamic cache proxylets also process the requests from the clients, getting the list of books from the server only in case of a cache miss. Normally, all requests would be handled by the server, increasing its CPU load. As the DPSs handle the requests, their CPU load becomes a little higher than that of the Squid servers, due to their increased processing responsibilities. However, that increase (up to 18%) is

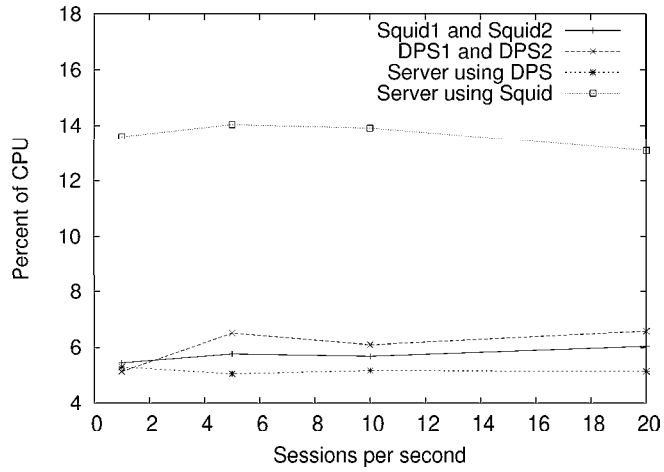

Figure 8: CPU of all components by sessions/s. Average CPU of the two Proxy Servers and Server CPU using Squid as proxy and DPS as proxy

easily compensated by the gains observed in terms of reduced response time, increased throughput and lower CPU load in the server. If we consider that the ALAN environment is implemented in Java and Squid is implemented in very optimized C code, the CPU load increase may be even considered smaller than one might expect.

8. Conclusion and Future Work

In this paper we presented a strategy to improve the quality of service of e-commerce sites and to make them more scalable. Our approach makes use of an application level active network as a deployment environment for an active cache-like solution used to distribute the processing of an e-commerce server over dynamic proxy servers. We evaluate the use of the ALAN architecture and the experimental results show benefits of the use of the dynamic cache proxylet. Our technique was able to relieve the e-commerce server, which may focus on other critical tasks. Response times were reduced by up to 70%, while the CPU load on the server was reduced by approximately 64%. Throughput also showed some improvement, although it was not as significant. Furthermore, we showed the viability of the use of ALAN to distribute e-commerce services and take the first step to use this architecture in real commercial applications.

As future work we intend first to evaluate different replacement policies for the e-commerce server and for the proxylets. After that, we intend to study how to extend the use of proxylets to other e-commerce operations, such as the selection of items and the purchase process, as well as to include techniques to improve consistency between proxylets and the server. Finally, we intend to consider other ways to increase the distribution of dynamic services using the ALAN environment, and possibly to test the solution in a realistic, large scale, geographically distributed setup using the PlanetLab infrastructure [PlanetLab, 2002].

References

Abrahao, B. and Benevenuto, F. (2003). Evaluating cache-layering to improve web cache system performance. In *Simpósio Brasileiro de Sistemas Multimídia e Web (Web-Midia)*.

Amir, E., McCanne, S., and Katz, R. H. (1998). An active service framework and its application to real-time multimedia transcoding. In *SIGCOMM*, pages 178–189.

Cao, P., Zhang, J., and Beach, K. (1998). Active cache: Caching dynamic contents on the web. In *Proc. of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 373–388.

Coutinho, B., Teodoro, G., Tavares, T., Pinto, R., Nogueira, D., Jr., W. M., and Guedes, D. (2002). Assessing the impact of distribution on e-business services. In *First Seminar on Advanced Research in Electronic Business*, Rio de Janeiro, RJ.

Fisher, M. (2001). D5: Active networking architecture. Technical report, The ANDROID Consortium.

Fry, M. and Ghosh, A. (1999). Application level active networking. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(7):655–667.

Gribble, S., Brewer, E., Hellerstein, J., and Culler, D. (2000). Scalable, distributed data structures for internet service construction.

Liabotis, I., Prnjat, O., and Sacks, L. (2001). Policy-based resource management for application level active networks.

Meira Jr., W., Cesário, M., Fonseca, R., and Ziviani, N. (1999). Integrating www caches and search engines. In *Proceedings of the IEEE 1999 Global Telecommunications Internet*, pages 1763–1769, Rio de Janeiro, RJ.

Meira Jr., W., Menascé, D., Almeida, V., and Fonseca, R. (2000). E-representative: a scalability scheme for e-commerce. In *Proceedings of the Second International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems (WECWIS'00)*, pages 168–175.

Menascé, D., Almeida, V., Riedi, R., Peligrinelli, F., Fonseca, R., and Jr, W. M. (2000). In search of invariants for e-business workloads. In *In Proceedings of the 2nd ACM e-Commerce Conference*, pages 56–65, Minneapolis, MN.

Mosberger, D. and Jin, T. (1998). httperf: A tool for measuring web server performance. In *Proc. of the Internet Server Performance Workshop*, pages 59–67.

PlanetLab (2002). An open platform for developing, deploying, and accessing planetary-scale services. Available at: `http://www.planet-lab.org/`.

Silva, R. D. (1998). Total management of transmissions for the end-user, a framework for user control of application behaviour.

Sun Microsystems, I. (1998). Java remote method invocation (rmi) specification. Available at: `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`.

Vingralek, R., Breitbart, Y., Sayal, M., and Scheuermann, P. (1999). Web++: A system for fast and reliable web service. In *Proceedings of 1999 USENIX Annual Technical Conference*, pages 171–184.

Wessels, D. and Claffy, K. (1997). Squid Internet Object Cache. Available at: `http://www.nlanr.net/squid/`.

Yoshikawa, C., Chun, B., Eastham, P., Vahdat, A., Anderson, T., and Culler, D. (1997). Using smart clients to build scalable services. In *Proceedings of the USENIX 1997 Annual Technical Conference*. USENIX Association.