A Hybrid Web Server Architecture for e-Commerce Applications

David Carrera, Vicenç Beltran, Jordi Torres and Eduard Ayguadé
{dcarrera, vbeltran, torres, eduard}@ac.upc.es

European Center for Parallelism of Barcelona (CEPBA)
Computer Architecture Department, Technical University of Catalonia (UPC)
C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6, E-08034
Barcelona (Spain)

Abstract

The performance of an e-commerce application can be measured according to technical metrics but also following business indicators. The revenue obtained by a commercial web application is directly related to the amount of clients that complete business transactions. In technical terms, a business transaction is completed when a web client successfully finishes a browsing session. In this paper we introduce a novel web server architecture that combines the best aspects of both the multithreaded and the event-driven architectures, the two major existing alternatives, to create a server model that offers an improved performance in terms of user session completions without loosing the natural ease of programming characteristic of the multithreading paradigm. We describe the implementation of this architecture on the Tomcat 5.5 server and evaluate its performance. The obtained results demonstrate the feasibility of the hybrid architecture and the performance benefits that this model introduces for e-commerce applications.

1. Introduction

Most e-commerce applications are distributed applications based on the well-known client/server paradigm that reside mostly in an application server and that are usually accessed by a remote thin web client. The communication protocol between the server and the client is the Hypertext Transfer Protocol (HTTP), which in the server side is parsed and processed by a component of the application server that is commonly known as the web container. Most generally, a web container can be considered as a web server that can support some language extensions to create more flexible web applications.

The performance of an e-commerce application can be measured in technical terms or following business indicators (see [8] for further discussion on this topic). Usually, the technical measurement units for a web server are replies per second (throughput) and the response time. In contrast, most performance metrics based on business indicators can be reduced to one: profit. The revenues a website can generate are directly related to the amount of commercial transactions the website can complete. In general, one commercial transaction is completed when user browsing session successfully finishes. Therefore, the web container architectures must be designed to obtain a high performance in user sessions units in order to be on the side of the interests of e-commerce applications.

The architectural design of most currently existing web containers follow the multithreading paradigm (Apache and Tomcat [7] are widely extended examples) because it leads to a natural ease of programming and simplifies the development of the web container logic. Unfortunately, this model is not especially indicated to obtain a high performance in terms of user session completions. Alternatively, an event-driven model (used in Flash[10] and in the SEDA[11] architecture) can be adopted to develop a web server container. This model solves some of the problems present in the multithreaded architecture but transforms the development of the web container into a hard task.

In this paper we introduce a new hybrid web server architecture that exploits the best of each one of the discussed server architectures. With this hybrid architecture, an event-driven model is applied to receive the incoming client requests. When a request is received, it is serviced following a multithreaded programming model, with the resulting simplification of the web container development associated to the multithreading paradigm. When the request processing is completed, the event-driven model is applied again to wait for the client to issue new requests. With this, the best of each model is combined and, as it is discusses in following sections, the performance of the server is remarkably increased in terms of user session completions.

The rest of the paper is structured as follows: section 2 discusses the characteristics of the two architectures involved in the creation of a hybrid web server, section 3 describes the implementation details of the hybrid web server evaluated on this paper, later, in section 4, we present the execution environment where the experimental results presented in this work were obtained. Finally, section 5 presents the experimental results obtained in the evaluation of the hybrid web server and section 6 gives some concluding remarks and discusses some of the future work lines derived from this work.

2. Web server architectures

There are multiple architectural options for a web server design, depending on the concurrency programming model chosen for the implementation. The two major alternatives are the multithreaded model and the event-driven model. In both models, the work tasks to be performed by the server are divided into work assignments that are assumed each one by a thread (a worker thread). If a multithreaded model is chosen, the unit of work that can be assigned to a worker thread is a client connection, which is achieved by creating a virtual association between the thread and the client connection that is not broken until the connection is closed. Alternatively, in an event driven model the work assignment unit is a client request, so there is no real association between a server thread and a client.

The multithreaded programming model leads to a very easy and natural way of programming a web server. The association of each thread with a client connection results in a comprehensive thread lifecycle, started with the arrival of a client connection request and finished with the connection close. This model is especially appropriate for short-lived client connections and with low inactivity periods, which is the scenario created by the use of non persistent HTTP/1.0 connections. A pure multithreaded web server architecture is generally composed by an acceptor thread and a pool of worker threads. The acceptor thread is in charge of accepting new incoming connections, after what each established connection is assigned to one thread of the workers pool, which will be responsible of processing all the requests issued by the corresponding web client.

The introduction of connection persistence in the HTTP protocol, already in the 1.0 version of the protocol but mainly with the arrival of HTTP/1.1, resulted in a dramatic performance impact for the existing multithreaded web servers. Persistent connections, which means connections that are kept alive by the client between two successive HTTP requests that in turn can be separated in time by several seconds of inactivity (think times), cause that many server threads can be retained by clients even when no requests are being issued and the thread keeps in idle state. The use of blocking I/O operations on the sockets is the cause of this performance degradation scenario. The situation can be solved increasing the number of threads available (which in turn results in a contention increase in the shared resources of the server that require exclusive access) or introducing an inactivity timeout for the established connections, that can be reduced as the server load is increased. When a server is put under a severe load, the effect of applying a shortened inactivity timeout to the clients lead to a virtual conversion of the HTTP/1.1 protocol into the older HTTP/1.0, with the consequent loss of the performance effects of the connection persistence.

In this model, the effect of closing client connections to free worker threads reduces the probability for a client to complete a session to nearly zero. It is especially important when the server is under overload conditions, where the inactivity timeout is dynamically decreased to the minimum possible in order to free worker threads as quickly as possible, which provokes that all the established connections are closed during think times. This causes a higher competition among clients trying to establish a connection with the server. If we extend it to the length of a user session, we obtain that the probability of finishing it successfully under this architecture is still much lower than the probability of establishing each one of the connections it is composed of, driving the server to obtain a really low performance in terms of session completions. This situation can be alleviated increasing the number of worker threads available in the server, but this measure also produces an important increase in the internal web container contention with the corresponding performance slowdown.

On the other hand, the event-driven architecture completely eliminates the use of blocking I/O operations for the worker threads, reducing their idle times to the minimum because no I/O operations are performed for a socket if no data is already available on it to be read. With this model, maintaining a big amount of clients connected to the server does not represent a problem because one thread will never be blocked waiting a client request. With this, the model detaches threads from client connections, and only associates threads to client requests, considering them as an independent work units. An example of web server based on this model is described in [10], and a general evaluation of the architecture can be found in [3].

In an event driven architecture, one thread is in charge of accepting new incoming connections. When the connection is accepted, the corresponding socket channel is registered in a channel selector where another thread (the request dispatcher) will wait for socket activity. Worker threads are only awakened when a client request is already available in the socket. When the request is completely processed and the reply has been successfully issued, the worker thread registers again the socket channel in the selector and gets free to be assigned to new received client requests. This operation model avoids worker threads to keep blocked in socket read operations during client think times and eliminates the need of introducing connection inactivity timeouts and their associated problems.

A remarkable characteristic of the event-driven architectures is that the number of active clients connected to the server is unbounded, so an admission control[4] policy must be implemented. Additionally, as the number of worker threads can be very low (one should be enough) the contention inside the web container can be reduce to the minimum.

Hybrid architecture

In this paper we propose a hybrid architecture that can take benefit of the strong points of both discussed architectures, the multithreaded and the event driven. In this hybrid architecture, the operation model of the event-driven architecture is used for the assignment of client requests to worker threads (instead of client connections) and the multithreaded model is used for the processing and service of client requests, where the worker threads will perform blocking I/O operations when required. This architecture can be used to decouple the management of active connections from the request processing and servicing activity of the worker threads. With this, the web container logic can be implemented following the multithreaded natural programming model and the management of connections can be done with the highest possible performance, without blocking I/O operations and reaching a maximum overlapping of the client think times with the processing of requests.

In this architecture the acceptor thread role is maintained as well as the request dispatcher role from the pure event-driven model and the worker thread pool (performing blocking I/O operations when necessary) from the pure multithreaded design. This makes possible for the hybrid model to avoid the need of closing connections to free worker threads without renouncing to the multithreading paradigm. In consequence, the hybrid architecture makes a better use of the characteristics introduced to the HTTP protocol in the 1.1 version, such as connection persistence, with the corresponding reduction in the number of client reconnections (and the corresponding bandwidth save). Additionally, and as it has been discussed for the event-driven architectures, some kind of admission control policy must be implemented in the server in order to maintain the system in an acceptable load level.

3. Hybrid architecture implementation

To validate the proposed hybrid architecture we have implemented it inside of Tomcat 5.5, a widely extended web container. Tomcat 5 is built in the top of the Java platform, which provides non blocking I/O facilities across different operating systems in its NIO (Non Blocking I/O) API. The implementation has been done modifying a pluggable component of the server generally named connector which is the server component in charge of handling communications with the client

Tomcat is an open-source servlet container developed under the Apache license. Its primary goal is to serve as a reference implementation of the Sun Servlet and JSP specifications, and to be a quality production servlet container. For the scope of this paper, two major components of Tomcat are considered: Coyote and Catalina. Coyote is the default Tomcat connector and deals with client connection request parsing and thread pooling. Catalina is a servlet container, and implements most of the web container logic. The implementation of the hybrid server architecture in Tomcat only affects Coyote.

Coyote follows a connection service schema where, at a given time, one thread (an HttpProcessor) is responsible of accepting a new incoming connection on the server listening port and assigning to it a socket structure. From this point, this HttpProcessor will be responsible of attending and parsing the received requests through the persistent connection established with the client, while another HttpProcessor will continue accepting new connections. HttpProcessors are commonly chosen from a pool of threads in order to avoid thread creation overheads. A connection timeout is programmed to close the connection if no more requests are received in a period of time. When a request is parsed, Coyote requires Catalina to process the request and to send the corresponding response to the client.

The implementation of the hybrid architecture changes the original Coyote threading and I/O model. One thread is in charge of accepting and registering, through a NIO selector, new incoming connections. When a registered connection becomes active (i.e. it has data available to read so a read operation over the socket will not be blocking), it is dispatched by the selector thread to small pool of HttpProcessor threads. Each HttpProcessor services only one request for each assigned active connection. The request is read and parsed, always without blocking the thread, and it is send to Catalina who processes the request. When the request is finished the connection is re-registered on the selector and the thread is sent back to the pool until a new active connection is assigned to it.

This implementation presents a major drawback when the system becomes overloaded because in this situation the acceptor thread allows new connections to enter the server faster than Catalina can service them, what results in a quick growth of the number of concurrent connections, which in turn causes a severe response time degradation. In consequence the number of client timeouts grows an the throughput decreases. To avoid this problem we have introduced a simple but effective admission control mechanism (similar to the backpressure technique described in [11]), that prevents the acceptor thread from accepting new connections while all HttpProcessors are busy.

4. Testing environment

The hardware platform for the experiments presented in this paper is composed of a 4-way Intel Xeon 1.4 GHz with 2GB RAM to run the web servers and a 2-way Intel XEON 2.4 GHz with 2 GB RAM to run the benchmark clients. For the benchmark applications that require the use of a database server, a 2-way Intel XEON 2.4 GHz with 2 GB RAM was used to run MySQL v4.0.18, with the MM.MySQL v3.0.8 JDBC driver. All the machines were running a Linux 2.6 kernel, and were connected through a switched Gbit network. The SDK 1.5 from Sun was used to develop and run the web servers.

The servers were tested in two different scenarios, one to evaluate the server performance for a static content application and another for a dynamic content environment. The workload for the experiments was generated using a workload generator and web performance measurement tool named Httperf[9]. This tool allows the creation of a continuous flow of HTTP requests issued from one or more client machines and processed by one server machine: the SUT (System Under Test). The configuration parameters of the benchmarking tool used for

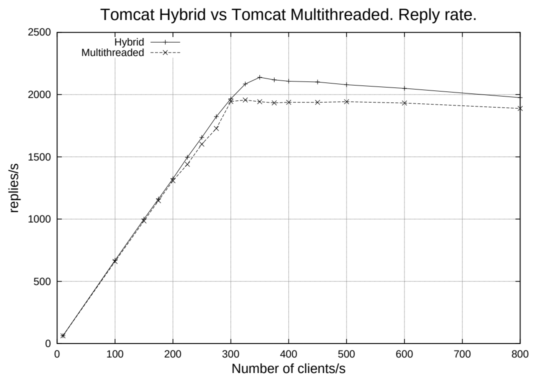

Figure 1. Throughput comparison under an static content workload

the experiments presented in this paper were set to create a realistic workload, with non-uniform reply sizes, and to sustain a continuous load on the server. One of the parameters of the tool represents the number of clients/s, i.e.the load. Each emulated client opens a session with the server. The session remains alive for a period of time, called session time, at the end of which the connection is closed. Each session is a persistent HTTP connection with the server, used by the client to repeatedly send requests, some of them pipelined. The requests issued by httperf were extracted from the surge[2] workload generator for the static content scenario and from the RUBiS [1] application for the dynamic content environment.

A static content application is characterized by the

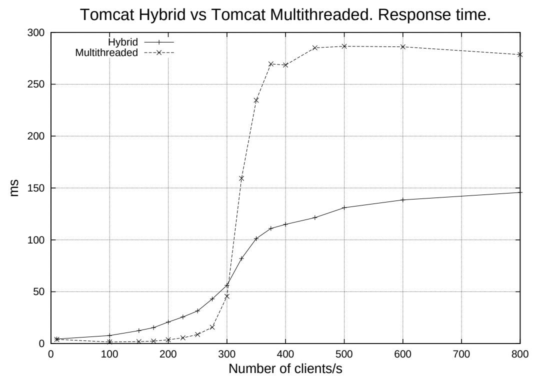

Figure 2. Response time under an static content workload

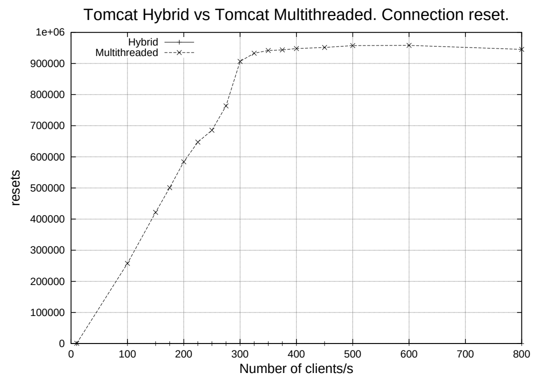

Figure 3. Number of connections closed by the server by a timeout expiration

short length of the user sessions as well as by the low computational cost of each request to be serviced. This is the scenario reproduced with the Surge[2] workload generator for these experiments. The request distribution produced by Surge is based on the observation of some real web server logs, from where it was extracted a data distribution model of the observed workload. This fact guaranties than the used workload for the experiments follows a realistic load distribution.

A dynamic content application is characterized by the long length of the user sessions (an average of 300 requests per session in front of the 6 requests per session for the static workload) as well as by the high computational cost of each request to be serviced (including embedded requests to external servers, such as databases). For our experiments, the chosen dynamic content benchmark application was RUBiS[1] (Rice University Bidding System), in its 1.4 servlets version. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. The workload distribution generated by Httperf was extracted from the RUBiS client emulator, which uses a Markov model to determine which subsequent link from the response to follow. Each emulated client waits for an amount of time, before initiating the next interaction, emulating on this way the ”thinking times observed in real clients. Httperf allows also configuring a client timeout. If this timeout is elapsed and no reply has been received from the server, the current session is discarded.

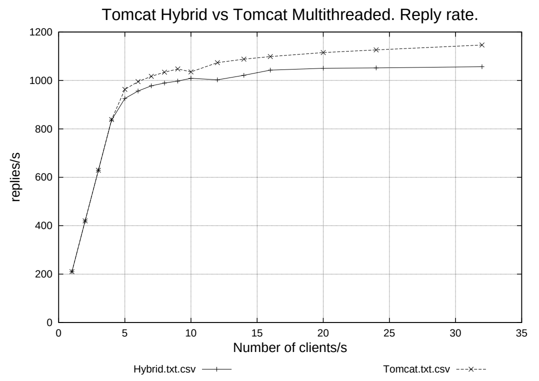

Figure 4. Reply throughput comparison under a dynamic content workload

5. Experimental results

In our experiments we evaluate which are the performance characteristics for each web server architecture, the original multithreaded Tomcat and our hybrid Tomcat implementation, for the two different scenarios already introduced in section 4: an static content application (Surge) and a dynamic content one (RUBiS). For the experiments, we configured Httperf setting the client timeout value to 10 seconds, and used the configuration described in section 4. Each individual benchmark execution had a fixed duration of 30 minutes for the dynamic content tests and 10 minutes for the static content experiments.

5.1. Static content

The first performance metric evaluated for this scenario is the obtained throughput for each architectural design, measured in replies per second. The results for both architectures are shown in figure 1. In the figures, it can be seen that the multithreaded architecture obtains a slightly lower performance than the hybrid one in terms of throughput, but the results are so close that they can be considered equivalent. It is remarkable that the same throughput is obtained by the hybrid architecture with a thread pool size of only 10 threads, while in the multithreaded architecture requires 500 threads to obtain the same result.

If we move from the throughput to the response time observed for each implementation, we can see that the hybrid architecture offers a clearly better result than the multithreaded one, as it can be seen in figure 2. When the system is not saturated (under a load equivalent to 300 new clients per second), the response time for the multithreaded server is slightly better than for the hybrid design, possibly because of the overhead introduced by the extra operations that the hybrid server must do to register the sockets in the selector and to switch them between blocking and non-blocking mode when moving from multithreaded to the event-driven and reverse. This effect would be dispelled in a WAN environment, where the latencies are much higher than in the Gbit LAN used for the experiments. When the system is saturated, beyond 300 new clients/s, the benefits of the hybrid architecture turn up and the response time reduction with respect to the multithreaded version of the server is of about a 50%, moving from a 300 ms average response time for the multithreaded architecture to a 150 ms response time for the hybrid design.

Another interesting difference between the behavior of the two studied architectures can be observed in figure 3. The figure shows the amount of connections that have been closed by the server because of a too long client inactivity period, causing the server timeout to expire and obligating the client to be reconnected to resume its session. As it can be seen, while the hybrid architecture is not producing this kind of situation (zero errors are detected by the benchmark client), a high number of errors are detected for the multithreaded architecture. This situation can be explained by the need of the multithreaded design to free worker threads to make them available for new incoming connections. This causes that the client inactivity periods must be avoided by closing the connection and requiring the client to resume its session with a new connection establishment when necessary. In the hybrid architecture, that assigns client requests as work units to the worker threads instead of client connections, this situation is naturally avoided and the cost of keeping a client connection event in periods of inactivity is equivalent to the cost of keeping the connection socket opened. The effect of this characteristic for the hybrid architecture is that all the reconnections are eliminated.

5.2. Dynamic content

Dynamic content applications implement a higher complexity and more developed semantics than static ones, which is usually translated into longer user sessions and involves that the common performance metrics are partially redefined in terms of business concepts. This means that e-commerce applications are more concerned about sales or business transactions than about more technical metrics such as the throughput or the response time offered by the server.

For this experiment, we consider that one of the most important metrics for an auction website (the scenario reproduced by RUBiS, see section 4 for more details) is the number of user sessions that are completed successfully. Each user that can complete its navigation session represents a potential bid for an item and in consequence a higher profit for the auction company.

Looking at figure 4, it can be seen that the throughput offered by both architectural designs is very similar, although the multithreaded architecture shows a slightly better performance when the server is saturated.

Looking at the number of sessions completed per second, in figure 5, it can be seen that the amount of successfully finished sessions reached by the hybrid architecture is clearly higher than the amount reached by the

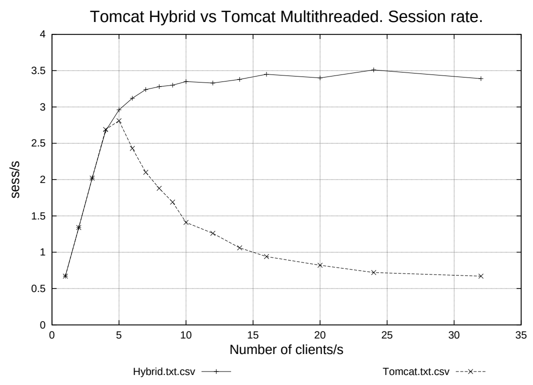

Figure 5. Successfully completed session rate under a dynamic content workload

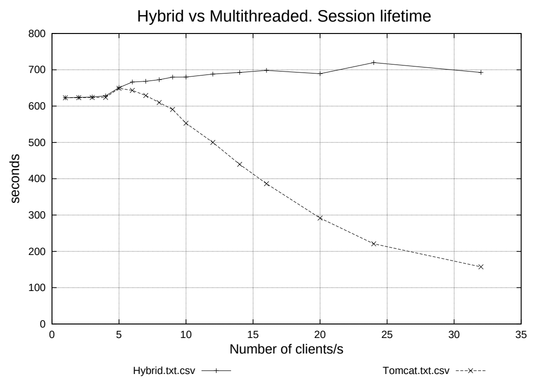

Figure 6. Lifetime comparison for the sessions completed successfully under a dynamic content workload

multithreaded design, especially beyond saturation. As it can be observed, the multithreaded architecture tends to decrease the number of completed sessions per second as the workload intensity is increased. This can be explained because under a pure multithreaded design the worker threads are obligated to close the client connections in order to be freed and rest available for new incoming connection requests. This situation, under high loads, leads to a scenario where the clients whose connections have been closed by the server, start experiencing problems to be reconnected because the amount of active clients trying to establish a connection is remarkably higher than the amount of worker threads disposed in the server. If this is extended to the amount of connections required to complete a long user session, characteristic of dynamic applications, the probability of being able to finish the session successfully is reduced to near zero, as it has already been discussed in section 2.

This explanation to the difference of performance between the two architectural designs in terms of session completion is supported by the results shown in figure 6 that indicate that the sessions completed by the multithreaded server are significantly shorter than the sessions completed by the hybrid server. This explains that the reply rate for the multithreaded server can be sustained even when the session rate is remarkably reduced because it proves that the sessions completed by the multithreaded server are those ones with less requests (the shorter ones). Completing only short sessions means that many active clients finishes their sessions unsuccessfully, which in turn may result in an important amount of unsatisfied clients that have received a very poor quality of service.

6. Conclusions and future work

In this paper we have shown how a web server hybrid architecture that combines the best of a multithreaded design with the best of an event-driven model can be used to obtain a high performance web server, especially when the throughput is measured in successfully completed user sessions per second. The proposed implementation into the Tomcat 5.5 code offers a slightly better performance than the original multithreaded Tomcat server when it is tested for a static content application, and a remarkable performance increase when it is compared for a dynamic content scenario, where each user session failure can be put into relation with business revenue losses. Additionally, the natural way of programming introduced by the multithreading paradigm can be maintained for most of the web container code.

Further research will be done toward the exploitation of the hybrid architecture benefits in the area of session based admission control, especially in presence of secure connections (SSL), where the cost of client reconnections can result in an enormous performance impact (see [5] for more details on this topic). The hybrid architecture also reduces the complexity of the tuning of a web container, which is an important step toward the implementation of autonomic servers[6].

Acknowledgments

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2004-07739-C02-01 and by the CEPBA-IBM Research agreement.

Please visit the Barcelona eDragon Research Group web page for additional information http://www.ciri.upc.es/eDragon

References

[1] C. Amza, A. Chanda, E. Cecchet, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks, 2002.

[2] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[3] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Evaluating the scalability of java event-driven web servers. In *2004 International Conference on Parallel Processing (ICPP'04)*, pages 134–142, 2004.

[4] H. Chen and P. Mohapatra. Session-based overload control in qos-aware web servers. In *INFOCOM*, 2002.

[5] J. Guitart, V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Characterizing secure dynamic web applications scalability. In *19th International Parallel and Distributed Symposium (IPDPS'05)*, 2005.

[6] IBM Research. *Autonomic computing. See http://www.research.ibm.com/autonomic*.

[7] Jakarta Project. Apache Software Foundation. *Tomcat. See http://jakarta.apache.org/tomcat*.

[8] A. Keller and H. Ludwig. Defining and monitoring service-level agreements for dynamic e-business. In *LISA*, pages 189–204, 2002.

[9] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[10] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[11] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.