Evaluating the Scalability of Java Event-Driven Web Servers

Vicenç Beltran, David Carrera, Jordi Torres and Eduard Ayguadé
{vbeltran, dcarrera, 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

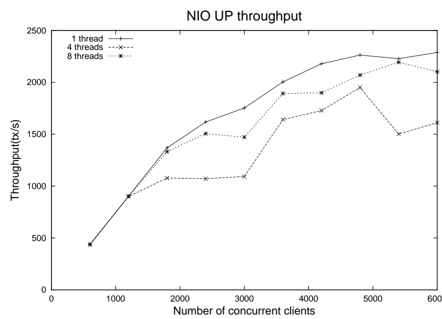
The two major strategies used to construct high-performance web servers are thread pools and event-driven architectures. The Java platform is commonly used in web environments but up to the moment it did not provide any standard API to implement event-driven architectures efficiently. The new 1.4 release of the J2SE introduces the NIO (New I/O) API to help in the development of event-driven I/O intensive applications. In this paper we evaluate the scalability that this API provides to the Java platform in the field of web servers, bringing together the majorly used commercial server (Apache) and one experimental server developed using the NIO API. We study the scalability of the NIO-based server as well as of its rival in a number of different scenarios, including uniprocessor, multiprocessor, bandwidth-bounded and CPU-bounded environments. The study concludes that the NIO API can be successfully used to create event-driven Java servers that can scale as well as the best of the commercial native-compiled web server, at a fraction of its complexity and using only one or two worker threads.

1 Introduction

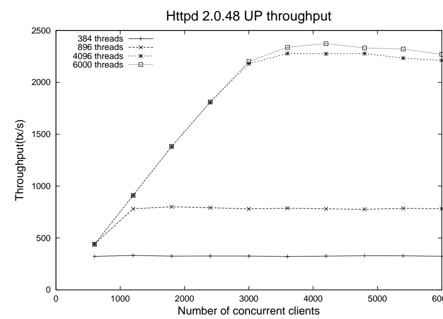
The development of Web Servers is always a challenging task because it implies the creation of high performance I/O strategies. Servers attending requests from thousands of clients simultaneously need to perform really efficiently if they want to offer a good Quality of Service. Two major strategies are commonly used to confront the service of requests in a Web Server. The first approach is based on a multithread or multiprocess strategy, assigning a different thread or process to the service of each incoming request. The second strategy, referred to as event-driven model, is based on the division of the request servicing process into a set of stages. One or more threads are in charge of each stage and make use of non-blocking I/O operations to respond simultaneously to a number of requests coming from different clients. At the moment, the multithreaded approach to the problem of attending to multiple concurrent client requests is widely used on commercial Web Servers.

The Java platform has been traditionally considered a low-performance environment to develop I/O intensive applications. Even on J2EE[10] environments, Java is only used as far as it is required by the standard and in general the entry point to a J2EE server, which is usually a Web Server, is delegated to native compiled applications. This is the case of many commercial J2EE Application Servers, such as the WebSphere Application Server[6] from IBM, which choose Apache[12] Web Server as its web entry point server. But this situation can change if the new API for efficient I/O operations included in the J2SE platform since its 1.4 release really comes to offer a high-performance I/O infrastructure. This new API, called NIO[11] (New I/O), provides the Java platform with a number of features traditionally available in most native compiled environments but still lacking in the the J2SE APIs. These features include the memory mapping of files and the availability of non-blocking I/O operations.

In this paper we evaluate the scalability of an experimental event-driven Java web server in a number of scenarios, bringing it together with the widely used Apache HTTP Server in our testing platform to compare their scalability on uniprocessor, multiprocessor, bandwidth-bounded and CPU-bounded environments. The obtained results demonstrate that the NIO API can be used to create Java web servers scaling (with the load and with the



(a) NIO



(b) httpd2

Figure 1. Throughput comparison on a uniprocessor (UP) system

number of available processors) as well as the best of the native-compiled commercial servers and at a fraction of their complexity and resource-consumption, independently of the execution scenario.

The use of event-driven architectures for web servers is an already explored area. Flash[9] is an asymmetric multi-process event-driven web server which exploits the creation of a set of helper processes to avoid thread-blocking in the main servicing process. Haboob[14] is also an event-driven web server, based on the concepts of staged event-driven servers introduced by SEDA[14]. JAWS[5] web server uses the Proactor[4] pattern to easily construct an event-driven concurrency mechanism. In [15], the authors propose a design framework to construct concurrent systems based on the combination of threading and event-driven mechanisms to achieve both high throughput and good parallelism. Some advanced topics about performance issues involving event-driven architectures have been studied in [7], [16] and [3]. The introduction of a non-blocking I/O API in the J2SE was preceded by the development of an extensively used API called NBIO[13] (Non-Blocking I/O), which was used to create the standard API NIO. None of the previously commented articles evaluate the causes of performance scalability, with respect to the workload intensity and the number of processors, of the event-driven architectures for Java web servers in comparison to the more commonly used multithreaded servers. This is the main contribution of this paper.

The remaining of the paper is as follows: section 2 remarks some important aspects about the operation and design of Web Servers and present some of the new features introduced in the Java platform with the Java NIO API. In section 3 we describe the execution environment used for these experiments. Next, in section 4 a performance evaluation of the studies web servers is performed for monoprocessor environments. Section 5 evaluates the nio web server and the Apache 2 web server on multiprocessors environments. Finally, section 6 give some concluding remarks derived from this work. An extended version of this paper containing more results can be found at [2].

2 Web Server architectures

The action of accepting new incoming requests on the server is usually performed by a thread which is always listening at a concrete port. When a new request arrives to the server, the request is read and then the server can mainly behave in two different ways depending on the underlaying I/O capabilities available on the execution environment: it can use one thread performing blocking I/O operations for each active socket or it can use one thread performing non-blocking I/O operation among all the active sockets. What really makes difference between these two possibilities is the fact that in the first case the thread which performs the blocking I/O operation is no longer available for processing new incoming requests until the service is completed, while in the second case the thread can continue attending other clients through other sockets. This difference of concept results in a completely different way of implementing servers: if the first version is chosen, a number of threads must be created in the server in order to give response to a high number of clients. If the second version is chosen, just one thread could (theoretically) attend a high number of clients simultaneously. Once a request have been read from the client socket, the web server has to provide the client with the data requested through the connection socket.

The 1.4 release of the Java 2 Standard Edition includes a new set of I/O capabilities created to improve the performance and scalability of intense I/O applications. The classic Java I/O mechanisms don't offer the

desired level of efficiency to run applications with high I/O requirements. Some widely existing I/O capabilities implemented on most of operating systems such as mapping files to memory, readiness selection of channels and non-blocking I/O operations, were not supported by previous versions of the J2SE and have been included in the 1.4 release. The NIO package includes a readiness selection operation on channels (SocketChannels and FileChannels). This select operation is used in the implementation of the main loop to detect which web clients have sent requests to the server or which sockets have more data available. The non-blocking features of the NIO API have been used to create a web server core, based on the event-driven strategy, which is is used in the scope of this paper to evaluate the capabilities of a server based on this technology. This experimental web server will be referred to as nio server from now.

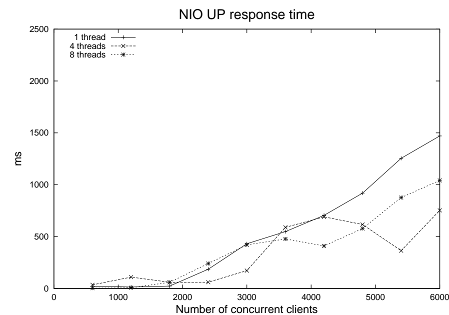
3 Testing environment

To perform the experiments, a 4-way 1.4Ghz Xeon machine with 2GB of memory was used to run the servers, and two 2-way 2.4Ghz Xeon system with 2GB of memory were used to run the workload generators. All them were running Linux as the operating system with a 2.6.2 kernel. The Java Runtime Environment (JRE) 1.4.1 from IBM was the chosen J2SE environment to run the Java servers. The commercial server chosen for the comparison is Apache 2.0.48 and it configured using a multithread schema instead of a multiprocess strategy.

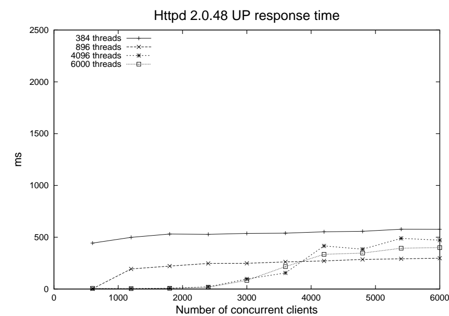
We used three different configurations for the network connection between the System under Test (SUT) and the workload generators. The first configuration connected the server machine to one client machine through a 100 Mbits ethernet interface. The second configuration connected each one of the two client machines to the SUT through a 100 Mbit/s ethernet interface, so we obtained an accumulated bandwidth between the server and the clients of 200 Mbit/s. Finally, the third configuration connected the SUT with a client machine through a 1 Gbit/s ethernet interface. Each connection between each client machine and the SUT used a crossed-link wire to avoid collisions.

3.1 Benchmark

The benchmark used to generate the workload for the experiments was Httperf[8]. This workload generator and web performance measurement tool allows the creation of a continuous flow of HTTP requests issued from one or more client machines and processed by one Web server machine: the SUT (System Under Test). The configuration parameters of the benchmarking tool used for 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 web server. One of the parameters of the tool represents the number of emulated clients on the client machine. Each emulated client issues a number of requests, some of them pipelined, over a persistent HTTP connection.



(a) NIO



(b) httpd2

Figure 2. Response time comparison on a uniprocessor (UP) system

The workload distribution generated by the Httperf benchmark was extracted from the Surge[1] workload generator. The distributions produced by Surge are 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.

4 Scalability on uniprocessors

The objective of this experiment was to compare the behavior of the NIO web server to the Apache 2 server. As a previous sub-objective of this task, we had to determine which configuration for each server, experimental or commercial, offered the best results so we could

compare the best configuration of each server. For this experiment the SMP support for the kernel of the SUT was disabled, so the system ran as an uniprocessor environment.

4.1 Configuration effects

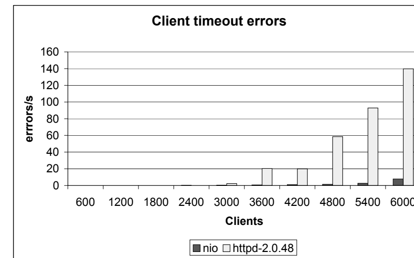
For this test, each server, the nio server and the httpd2 (Apache 2.0.48), was tested under a number of settings. From these configurations it was derived which one produced the best performance for each Web Server. The nio server was tested with 1, 4 and 8 worker threads and the httpd2 server was configured with 500, 1000, 4000 and 6000 threads. Each test was run for a time period of 5 minutes, with the Httperf benchmark configured to produce a constant workload intensity equivalent to having a range of 600 to 6000 concurrent clients, and each connected client producing an average of 6,5 requests grouped in a session. For this first experiment, the workload was entirely produced on one client machine connected to the SUT through the 1 Gbit/s link, so it was ensured that the test was not bandwidth-bounded.

The obtained throughput for each server can be seen in figure 1. The response time for the nio and httpd2 servers can be seen in figure 2.

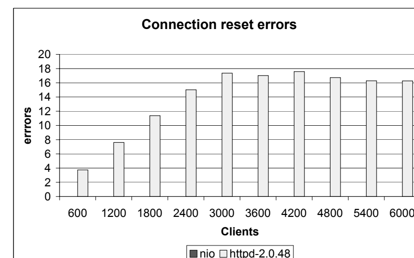
Observing the throughput results in figure 1, httpd2 seems to scale better than nio when the load increases. Its throughput raises linearly respect to the intensity of the generated workload whereas the obtained results for the nio server indicate that on a uniprocessor system it has more problems to be adapted to the increasing workload intensity.

Concerning to the configurations of the servers, it is remarkable for the httpd2 server that the size for the thread pool offering a best result for this experiment was 4096. According to our experiments, configuring the httpd2 server with a pool size of 6000 threads offered a slight performance increase with respect to the 4096 threads configuration but reduced significantly the stability of the system, even hanging the system several times. About the nio configurations, it is specially noticeable that with only one or two worker threads (with an additional acceptor thread) it can achieve the same throughput than the httpd2 server with 4096 threads. This characteristic of the nio server is specially important on a Java server, which is commonly limited to spawn a maximum of 1000 threads for the JVM.

For the test, it was ensured that the network bandwith was not a limiting factor because the observed bandwidth usage was always under 40MB/s (results on bandwidth usage can be found in [2]), which is less than the maximum achievable bandwidth for a 1 Gbit link, which was the used connection between the SUT and the client.



(a) Client timeout



(b) Connection reset

Figure 3. Connection errors

The client is not the bounding factor on this test either, because as it will be seen in section 5, higher throughputs are obtained when running the same tests with the SMP support activated in the kernel. As it was expected, there is a linear relation between the achieved throughput of each server and the bandwidth required by each test.

4.2 Considerations about connection errors, throughput and response time

The response time observed for the httpd2 server, as it can be seen in figure 2, was surprisingly low in average in comparison with the results obtained for the nio server for all the configurations. As it seemed a suspicious result, we studied the number of connection errors detected on the benchmark client. At first glance it seemed that with the httpd2 server a higher amount of connection errors were detected on the client machine. These errors can be divided on client timeouts and connection resets.

A client timeout error happens when the socket timeout defined by the client expires. This timeout value is used when establishing a TCP connection, when sending a request, when waiting for a reply, and when receiving a reply. If during any of those activities a request fails to make forward progress within the alloted time, httperf

considers the request to have died, closes the associated connection or session and increases the client-timeout error count. For our tests, the emulated clients established a 10 seconds socket timeout value.

A connection reset error takes place when a server socket timeout expires, and it is seen from the client side of the connection as an unexpected socket close. Typically, it is detected when the client attempts to send data to the server at a time the server has already closed its end of the connection.

The results for both client timeout and connection reset errors observed in the nio and httpd2 experiments can be seen in figure 3. They are expressed in number of errors per second. An interesting conclusion about connection reset errors can be extracted from figure 3(b). On it, it can be seen that while the nio server never produces connection reset errors, the amount of this kind of errors observed for the httpd2 server is not negligible and in a phase of the test even increases linearly with respect to the workload intensity. The cause for this difference is that because the multithreaded architecture of httpd2 binds one client connection to one server thread, it needs to free threads from their assigned clients to be able to process new client connections. This forces the server to automatically disconnect clients after an inactivity period. The inactivity time before closing a connection is the server socket timeout and is usually set to low values, depending on the expected workload intensity. For our experiments, we set up htppd2 with a timeout value of 15 seconds. Each client emulated by the workload generator, httperf, produced a workload based on the alternation of activity periods and think time periods. When the think time of a client exceeds the server timeout, a connection reset error takes place. As far as the nio server doesn't associate connected clients with server threads, it does not need to apply disconnection policies to its clients.

With respect to the client timeout errors, it can be observed that the number of errors of this type present in the tests for the httpd2 server is much higher than for the nio server. The explanation to this fact originates from the way clients are attended. In a multithreaded server, such as httpd2, each thread is binded to a client connection. When a request is received, the requested resource is located and a blocking I/O operation on the client socket is performed. This operation is not finished upon all the response is sent to the client. This virtually sequences the way in which responses are sent to clients and propitiates that waiting clients see their timeouts expire. On the other hand, in an event-driven server, such as nio, one or more worker threads attend a number of clients each one but never perform blocking operations on the socket sharing in a more fair way the network resource between clients. Individual responses are not sent as fast as with a multithreaded server (i.e. they obtain higher response times) but socket inactivities are less usual, avoiding timeouts to expire. When a socket send buffer is full, the in course operation gets blocked and the nio server starts attending another connected client which has room in its socket buffer for the data to be sent. This effect can be observed in figures 2(b) and 2(a).

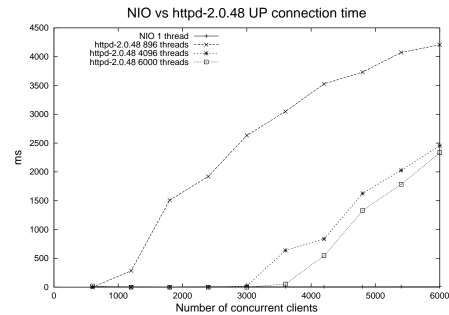


Figure 4. Connection time for the best configuration of nio and httpd2

Another noticeable fact is that the event-driven architecture makes possible to reach shorter waiting times for each connection to be established, as it can be seen in figure 4. This situation is explained because the multithreaded schemes use one thread for each client, and it implies that when the number of simultaneous clients exceeds the number of available threads for the server, contention to access the web server appears and provokes a fast increase of the time needed to establish connection with the server. In the scope of an event-driven server, the number of concurrent clients has nothing to do with the number of worker threads running on it and all clients can be attended without causing contention. The connection time value for the nio server on our experiments has been always below 10 ms. It is also a matter of fact that for the httpd server the point at which the connection time starts growing exponentially is not always directly related to the size of the pool of threads. When it is set up with a pool size of 896 threads, the connection time starts degrading when the workload intensity exceeds the maximum number of available concurrent connections for the server (i.e. the number of threads). On the other hand, when the overload introduced in the system caused by the costs of managing higher pool sizes (4096 or 6000 threads) causes that the connection time starts degrading before reaching the maximum number of available concurrent

connections for the server. The high load provoked by the workload intensity and the overhead caused by the management of a high number of threads present in the system is what will finally limit the maximum achievable throughput for a multithreaded server.

4.3 Performance effects of limiting factors

With this experiment we wanted to study the two servers in different conditions: when the system was bandwidth-bounded and when it was CPU-bounded. As it was said in section 4.1, the SUT kernel was configured without SMP support. For the client machines, three configurations were used: one client with a 100Mbit/s link, two clients with a 100Mbit/s link each one and one client with a 1 Gbit/s link.

The throughput observed in this experiment for the three network configurations can be seen in figure 5, and the resulting response times are shown in figure 6. When the most limiting factor in the system is bandwidth availability (in the 100 and 200 Mbit/s configurations, as it is presented in [2]), both httpd2 and nio servers have similar behaviors. Their throughput scales linearly with the workload intensity up to the moment when the maximum available bandwidth is reached. In that point, nio advances the httpd2 server. This difference is hardly appreciable with a 100 Mbit/s bandwidth, and more obvious when it is increased to 200 Mbit/s. The reason of the better performance of the nio server with respect to the httpd2 server is that the httpd2 server is obligated to apply a socket activity timeout to its clients to sustain an acceptable quality of service, and it is translated to the ethernet level as an increase in the experienced network congestion. As it is shown in section 4.2, the number of connection reset errors is considerable for the httpd2 server and inexistent for the nio server. This additional network load for the httpd2 server is more obvious as higher is the limiting bandwidth, and loses relevance when the computing capacity of the system becomes the limiting factor for the web server. On the other hand, when the bottleneck is the computing capacity of the system, the httpd2 server scales better with the workload intensity up to the moment when the overhead of rejecting a huge number of connections per second starts degrading its performance, moment at which the nio server advances it with a higher observed throughput.

Observing the obtained response times in figure 6, it can be seen that when the limiting factor in the system is the bandwidth, the response time for both servers is very close because it is determined by the network capacity. When the bottleneck is the CPU (in the 1 Gbit/s connection), the response time observed for the servers is clearly different. The nio server response time, as explained in section 4.2, increases with the workload intensity because thanks to its event-driven architecture all connected clients are attended concurrently without allowing the client connection timeouts to expire. This forces the server to dedicate a part of the bandwidth to each client and enlarging on this way the response time for all them. The httpd2 server responds to one client at a time, seen from the network viewpoint, and it reduces the response time for successful connections (those whose timeout does not expire). As far as the httperf benchmark only takes into account successful connections, the observed average response time for the server is kept low.

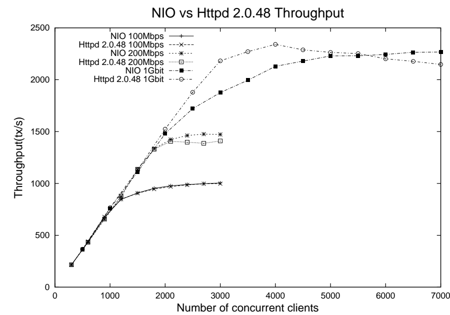


Figure 5. Throughput scalability on a uniprocessor (UP)

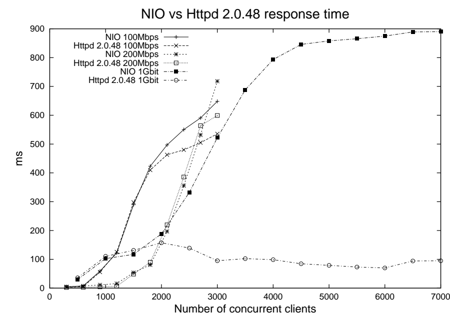


Figure 6. Response time scalability on a uniprocessor (UP)

5 Scalability on multiprocessors

Once the scalability for each web server under different workload intensities was determined with the experiments presented on section 4, it was time to explore how well each server could scale with respect to the number

of available processors in the system. The procedure for the experiment was analogous to the followed one on section 4. First of all, some different configurations for each tested web server were studied on the SUT, now configured with full SMP support (4 processors available on the system). After that, the most performing-configuration observed for httpd2 web server and the most performing-configuration of the nio-server were brought together on a SMP environment with the objective of comparing their scalability properties.

5.1 Best-performing configurations for a SMP system

For this test, each server was tested under a number of settings to evaluate their performance trends on a multiprocessor environment. The nio server was tested with 2, 3 and 4 worker threads and the httpd2 server was configured with 2000, 4000 and 6000 threads. Each test was run for a time period of 5 minutes, with the Httperf benchmark configured to produce a constant workload intensity equivalent to having a range of connected clients from 600 to 6000, and each client producing an average of 6,5 requests grouped in a session. The workload was entirely produced on one client machine connected to the SUT through the 1 Gbit/s link. So it was ensured that the test was not bandwidth-bounded (more results on bandwidth usage can be found in [2]).

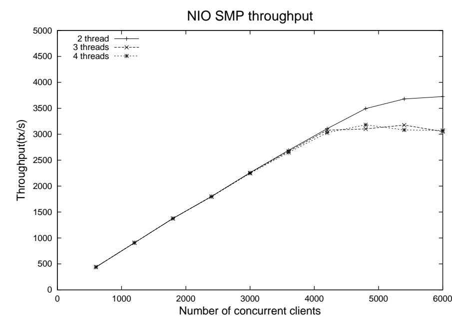
The obtained throughput for each server can be seen in figure 7. The response time for the nio and httpd2 servers can be seen in figure 8. Observing at the obtained results for the nio and httpd2 servers' throughput, it can be seen that their behavior is very similar. The httpd2 server gets higher throughputs than nio, but the difference is pretty short.

The response time observed for the nio server is worse than the results obtained for the httpd2 server, but it could be explained again by the connection errors introduced by httpd2, as it has been explained in section 4.2. The obtained values for both nio and httpd2 are shown in figure 8.

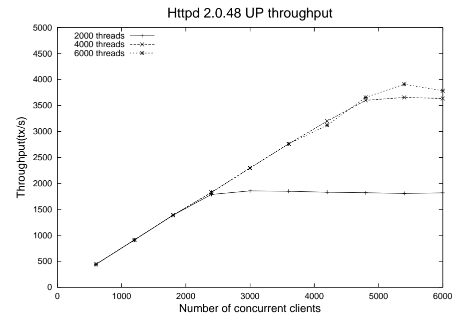
The best configuration for the nio server is that one involving two worker threads. For a uniprocessor kernel, best results were obtained with one simple worker thread, so the availability of more processors on the system favors the use of more worker threads attending clients in parallel.

5.2 Scaling with the number of processors

The objective of the following experiment was to explore the scalability properties of the two servers when the execution environment is moved from an uniprocessor system to a multiprocessor.



(a) NIO

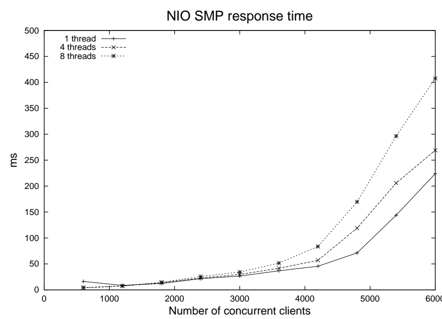


(b) httpd2

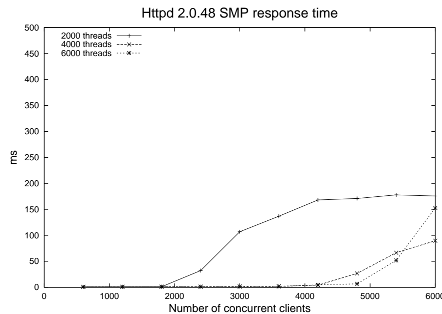
Figure 7. Throughput comparison on a 4-way SMP system

For the tests, we used one client machine connected to the SUT through the 1Gbit/s link. For 1-CPU configurations, a kernel without SMP support was used. For the SMP experiments, a kernel supporting the 4-way SMP system was used, so 4 processors were available in the system. The results for the uniprocessor (UP) system are obtained from the experiments presented on section 4.3. The servers' settings for the uniprocessor and multiprocessor tests were extracted from results obtained on section 4.1 and 5.1 respectively. Thus, the best settings for each environment were used, resulting in 2 worker threads for the nio server and 4096 threads for the http2 server.

The obtained results reveal that the nio server scales with the number of processors as well as the native-compiled and multithreaded httpd2 server for the response time, shown in figure 10, and for the throughput, shown in figure 9. The throughput obtained by both servers on the SMP environment doubles the value obtained on the uniprocessor when it is stabilized. The values reached by the nio and httpd2 servers are equivalents and can be considered to be in the same range of

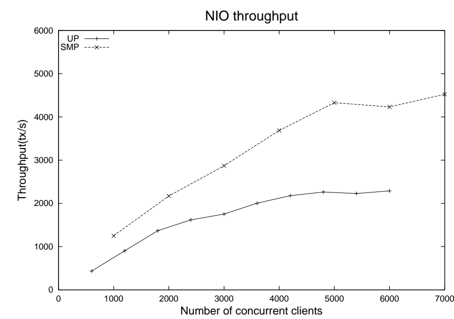


(a) NIO

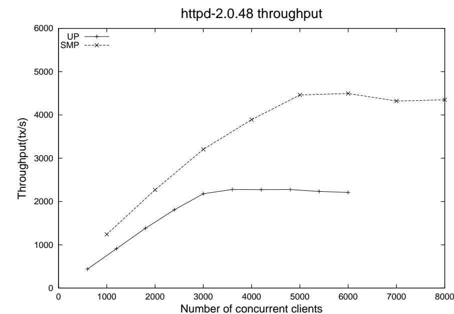


(b) httpd2

Figure 8. Response time comparison on a 4-way SMP system



(a) NIO throughput



(b) httpd-2.0.48 throughput

Figure 9. Throughput scalability from 1 to 4 CPUs

results. The observed response time for the two servers on the SMP environment is significantly lower than the obtained values for the uniprocessor configuration.

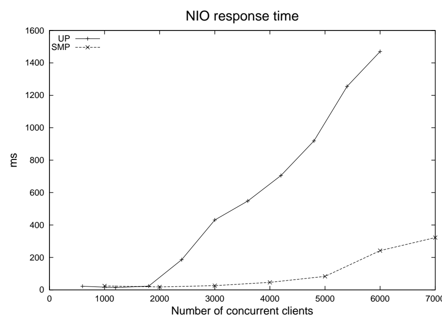
6 Conclusions and Future Work

In this paper we have presented the potential benefits of using event-driven architectures on web servers. Concretely, we have studied how the new non-blocking I/O API (NIO) included in the J2SE platform since its 1.4 release can help in the development of high-performance event-driven Java web servers. It has been probed that our experimental NIO-based server can outperform a widely used commercial native-compiled web server but reducing the complexity of the code and the system resources required to run it.

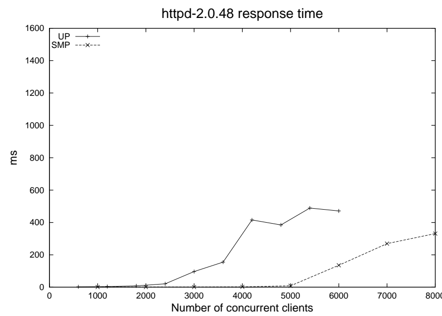
The NIO API applied to the creation of event-driven Web Servers scales well when changing the workload intensity and when changing the number of processors and it makes it at least as well as the Apache 2 server.

Choosing a Java event-driven architecture to implement next-generations high-performance web servers can be an option, but the really interesting conclusion of this study is that Java Application Servers can no more need to use native-compiled Web Servers as their web interfaces. Many commercial application servers choose Apache as their web interface because it is supposed to offer a better performance than Java Web Servers. But these decisions compromises the portability of the entire middleware. Choosing a Java event-driven architecture as the web interface for a Java Application Server can facilitate the integration of both elements, Web Server and Application Server, without paying the price of low performance.

A derived conclusion obtained from the results presented in this paper is that the event-driven architecture can be widely exploited on multiprocessors. Although it is an already proposed idea in the literature referring to staged servers, it has not been applied to application servers yet. Dividing the server in pipelined stages, adding one or more threads to each stage and assigning a processor affinity to each thread can convert a multiprocessor running an staged event-driven Java application server in a real high-scalable request processing pipeline.



(a) NIO Response time



(b) httpd-2.0.48 response time

Figure 10. Response time scalability from 1 to 4 CPUs

Acknowledgments

We are grateful to our colleague Josep Solé-Pareta and to his team, whose support in the configuration of the network for the testing platform has made possible this work. This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01 and by the CEPBA-IBM Research agreement.

References

[1] 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.

[2] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. Evaluating the new java 1.4 nio api for web servers. Technical Report UPC-DAC-2004-18 / UPC-CEPBA-2004-4, Technical University of Catalonia (UPC), 2004.

[3] S. Harizopoulos and A. Ailamaki. Affinity scheduling in staged server architectures. Technical Report CMU-CS-02-113, Carnegie Mellon University, 2002.

[4] J. Hu, I. Pyarali, and D. Schmidt. Applying the proactor pattern to high-performance web servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems. IASTED*, October 1998.

[5] J. Hu and D. Schmidt. *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. Wiley & Sons, 1999.

[6] IBM Corporation. WebSphere Application Server. http://www.ibm.com/websphere.

[7] J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance (extended abstract). In *LCTES/OM*, pages 182–187, 2001.

[8] 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.

[9] 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.

[10] Sun Microsystems, Inc. Java 2 Platform, Enterprise Edition (J2EE). http://java.sun.com/j2ee.

[11] Sun Microsystems, Inc. *New I/O APIs. 2002. http://java.sun.com/j2se/1.4.2/docs/guide/nio*.

[12] The Apache Software Foundation. *Apache HTTP Server Project. http://httpd.apache.org*.

[13] M. Welsh. *NBIO: Nonblocking I/O for Java. http://www.eecs.harvard.edu/ mdw/proj/java-nbio*.

[14] 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.

[15] M. Welsh, S. Gribble, E. Brewer, and D. Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, UC Berkeley, April 2000.

[16] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.