Performance and scalability of EJB applications

Emmanuel Cecchet, Julie Marguerite and Willy Zwaenepoel
Rice University
{cecchet, margueri, willy}@rice.edu

ABSTRACT

We investigate the combined effect of application implementation method, container design, and efficiency of communication layers on the performance scalability of J2EE application servers by detailed measurement and profiling of an auction site server.

We have implemented three versions of the auction site. The first version uses stateless session beans with bean-managed persistence, making only minimal use of the services provided by the Enterprise JavaBeans (EJB) container. The second version makes extensive use of the container services using entity beans with container-managed persistence. The third version applies the session façade pattern, using session beans as a façade to access entity beans. We evaluate these different implementations on two popular open- source EJB containers with orthogonal designs. JBoss uses dynamic proxies to generate the container classes at run time, making an extensive use of reflection. JOnAS pre-complies classes during deployment, minimizing the use of reflection at run time. We also evaluate the communication optimizations provided by each of these EJB containers.

The most important factor in determining performance is the implementation method. EJB applications with session beans perform as well as a Java servlet implementation and an order-of-magnitude better than most of the implementations based on entity beans. Use of session façade beans improves performance for entity beans, but *only* if local communication is very efficient. Otherwise, session façade beans degrade performance.

For the implementation using session beans, communication cost forms the major component of the execution time on the EJB server. The design of the container has little effect on performance. For implementations using session façade beans, local communication cost is critically important. With entity beans, the design of the container becomes important as well. In particular, the cost of reflection affects performance.

Keywords

EJB container design, performance, scalability, communication optimization, profiling.

1. INTRODUCTION

As the popularity of dynamic-content Web sites increases rapidly, there is a need for maintainable, reliable, available, secure and above all scalable platforms to host those sites. The Java™ 2 Platform Enterprise Edition (J2EE) specification is Sun's solution to address these issues. J2EE has been primarily targeted at n-tier application development [2]. It defines a set of Java APIs to build applications, and provides a runtime infrastructure for hosting applications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Four different containers are defined in the J2EE specification to provide a runtime for application components as depicted in figure 1. The Enterprise JavaBeans (EJB) server is often the bottleneck in J2EE applications [7]. This paper seeks to explain the effect of application implementation methods, container design, and efficiency of communication layers on the performance of an EJB server and the overall application. We have developed three different EJB implementations of an auction site modeled after eBay.com [11]. The semantics are the same for each implementation of the application.

We use three different implementation methods: stateless session beans, entity beans, and entity beans with session façade beans. For further comparison, we have also implemented a Java servlet version of the application that does not use EJB.

We evaluate two orthogonal container designs that are representative of most EJB containers available at this time. The dynamic proxy approach, used in the popular JBoss [13] open-source EJB server, generates the container classes at run time, making extensive use of reflection. Most commercial implementations and the JOnAS [15] open source EJB container use pre-compilation: classes are generated during deployment, reducing the use of reflection at run time. Reflection is known to be slow, but it is often claimed that its cost is masked by network latency or database accesses [1]. We will, however, show that reflection can be an important factor in determining performance. We also configure the EJB servers with and without communication optimizations.

We use open-source software in common use for our experiments: the Apache Web server [5], the Tomcat servlet server [12], the JBoss [13] and JOnAS [15] EJB servers and the MySQL [16] relational database. We have posted all software, configuration files, and full experiment reports on our web site http://www.cs.rice.edu/CS/Systems/DynaServer to allow others to reproduce the results and evaluate the impact of new designs on performance and scalability.

Each server runs on a separate node. In all cases except one, the processor on the EJB server is the bottleneck. The memory and disk are never a limiting resource. The network can reach very high utilization when few services from the EJB container are used.

The most important factor in determining performance is the implementation method. EJB applications with session beans perform as well as a Java servlet implementation and an order-of-magnitude better than most of the implementations based on

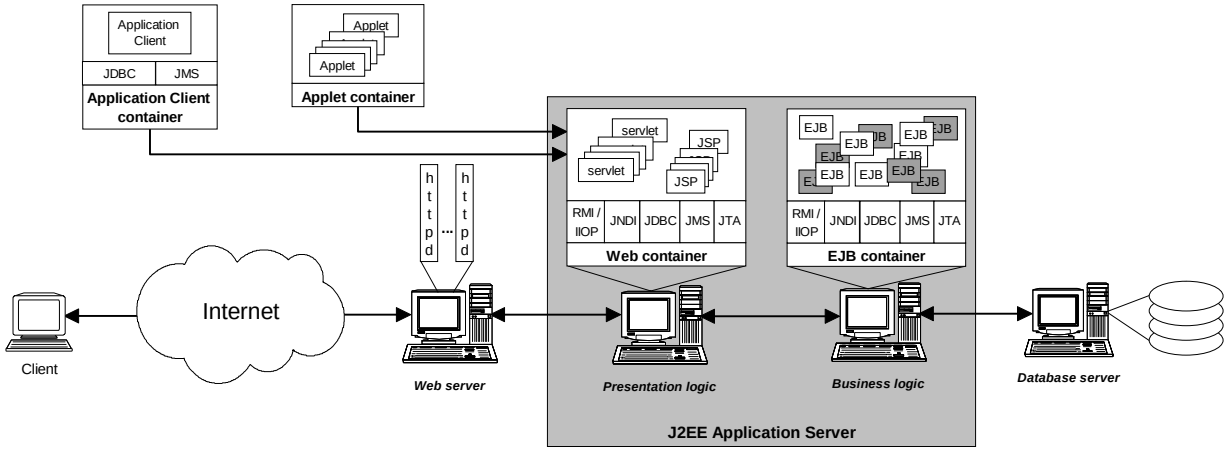

Figure 1. Enterprise Java Beans integration in the J2EE framework

entity beans. Use of session façade beans improves performance, but only if local communication is very efficient.

For the implementation using session beans, communication cost forms the major component of the execution time on the EJB server. The design of the container has little effect on performance. For implementations using session façade beans, local communication cost is critically important. With entity beans, the design of the container becomes important as well. In particular, the cost of reflection affects performance.

The outline of the rest of this paper is as follows. Section 2 provides some background on EJB. Section 3 provides detailed description of the alternative implementation methods, container designs, and communication optimizations. Section 4 describes the auction site and provides some complexity measures for the various implementation methods. Section 5 presents our experimental environment and our measurement methodology. Section 6 discusses the results of our experiments. Related work is presented in Section7. Section 8 concludes the paper.

2. BACKGROUND

An EJB server provides a number of services such as database access (JDBC), transactions (JTA), messaging (JMS), naming (JNDI) and management support (JMX). The EJB server manages one or more EJB containers. The container is responsible for providing component pooling and lifecycle management, client session management, database connection pooling, persistence, transaction management, authentication and access control.

In this paper, we consider two types of EJB: entity beans that map data stored in the database (usually one entity bean instance per database table row), and session beans that are used to perform temporary operations (stateless session beans) or represent temporary objects (stateful session beans).

A bean developer can choose to manage the persistence in the bean (Bean Managed Persistence or BMP) or let the container manage the persistence (Container Managed Persistence or CMP). In the latter case, a deployment descriptor contains a one-to-one mapping between bean instance variables and database columns. The container uses the descriptor to generate the necessary SQL statements and ensure concurrency control on the database. With BMP beans the programmer embeds the SQL queries in the bean code and only uses the database connection pooling and transaction management services of the container.

3. DESIGN ALTERNATIVES

3.1 Implementation Methods

We implement a servlet version and three EJB versions. The servlet version implements both the business logic and the presentation logic in the servlets in the usual manner. We next describe the three EJB versions.

3.1.1 Session beans with BMP

We use session beans with bean-managed persistence (SB BMP) to implement the business logic, leaving only the presentation logic in the servlets as depicted in figure 2.

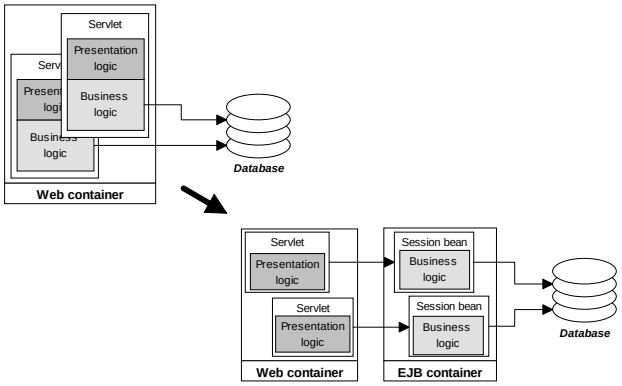

Figure 2. Session bean implementation

This implementation uses the fewest services from the EJB container. The session beans benefit from the connection pooling and the transaction management provided by the EJB server. It greatly simplifies the servlet code, in which the connection pooling has to be implemented by hand.

3.1.2 DAO separation with EB CMP

In this implementation, we extract the data access code from the servlets, and move it into Data Access Objects that we implement using entity beans with container-managed persistence (EB

CMP). The business logic embedded in the servlets directly invokes methods on the entity beans that map the data stored in the database (see top part of figure 3). With CMP, the vast majority of the SQL queries is generated by the EJB container. EJB 1.1 CMP, however, requires stateless session beans to execute complex queries involving joins on multiple tables. To avoid fine-grain access of getter/setter methods of the beans, we provide functions that return results populated with the values of the bean instance attributes.

The goal of this implementation is to evaluate the cost of the container's persistence service and the impact of fine-grain accesses between the Web and EJB containers.

3.1.3 Session façade

The session façade pattern [3] uses stateless session beans as a façade to abstract the entity components as shown in figure 3. It reduces the number of business objects that are exposed to the client over the network, thus providing a uniform coarse-grained service access layer. Calls between façade and entity beans are local to the EJB server and can be optimized to reduce the overhead of multiple network calls (see Section 3.3).

This implementation involves a larger number of beans, and thus stresses the component pooling of the container. It also exploits the database connection pooling, transaction manager and persistence services.

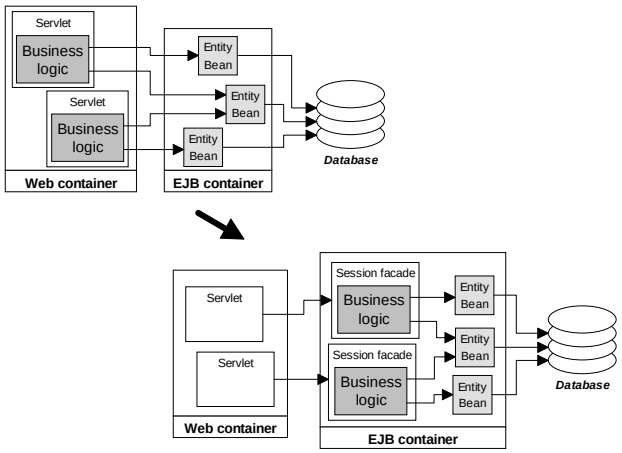

Figure 3. Refactoring the Entity Beans design with a session façade design pattern

3.2 EJB container design

An EJB container is a component that provides the EJB services to a particular EJB. It acts as an interface between the client and the bean. In fact, the client only interacts with the home and component interfaces that are provided by the container and forward the calls to the bean. So, each bean access is done through container-generated classes. There are two main approaches to design an EJB container, differing in how and when it generates those classes. With the pre-compiled approach, container classes are compiled at deployment time. The other method uses dynamic proxies to generate the classes at run time.

3.2.1 Pre-compiled approach

In a precompiled approach, the container generates custom implementations of the home and component interfaces so that it can directly call the appropriate method of the bean instance. The resulting classes have to be available for the client by way of the classpath or the ejb-jar file. This is the approach used in most commercial EJB containers and in the JOnAS EJB container.

The container vendor provides a tool to generate the container classes. The tool provided with JOnAS is called GenIC. GenIC generates the source for the container classes for all the beans defined in the deployment descriptor, and compiles them using the Java compiler. Then, it generates stubs and skeletons for those remote objects using the RMI compiler. Finally, it adds the resulting classes in the ejb-jar file if needed.

3.2.2 Dynamic proxy based container

With this approach, the container uses dynamic proxy technology [17] to generate the home and component interfaces at run time. A dynamic proxy is an object generated at run time that implements some specified interfaces and is responsible for routing the calls using reflection. Using reflection the proxy can map method signatures to the corresponding implementations or locate a bean given the name of the class. The client sends its calls to the proxy that analyzes and forwards them to the bean using reflection.

In the JBoss container, home and object interfaces are constructed as dynamic proxies. They use four types of proxy classes: one for the home interface and three for the component interface according to the type of the bean (entity, stateless session, stateful session)

3.3 Communication layer

Remote Method Invocation (RMI) is the object request broker (ORB) used by EJB. JBoss relies on Sun's RMI using JRMP (Java Remote Method Protocol) on top of TCP/IP, but it uses a specific registry and naming called JNP (Java Naming Provider) providing hierarchical namespaces. JOnAS can use either Sun's RMI or a modular ORB called Jonathan [10]. Jonathan has an RMI personality called Jeremie. Jeremie uses a different protocol, GIOP (General Inter-ORB Protocol), and can also optimize local communication.

To reduce the cost of marshalling, JBoss offers an optimization that passes objects by reference instead of by value. Although not compliant to the specification, this optimization is commonly used and it is the default setting in JBoss. Jeremie also uses this technique for local calls.

4. APPLICATION

The RUBiS (Rice University Bidding System) models an auction site. Its design aims to reduce as much as possible the load on the database server, allowing us to saturate the middle-tier and determine its maximum throughput and scalability.

4.1 Description

Our auction site defines 26 interactions that can be performed from the client's Web browser. Among the most important ones are browsing items by category or region, bidding, buying or selling items, leaving comments on other users and consulting one's own user page (known as myEbay on eBay [11]). Browsing items also includes consulting the bid history and the seller's information. We define two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions.

We sized our system according to observations found on the eBay Web site. We always have about 33,000 items for sale, distributed

among eBay's 40 categories and 62 regions. We keep a history of 500,000 past auctions. There is an average of 10 bids per item, or 330,000 entries in the bids table. The users table has 1 million entries. We assume that users give feedback (comments) for 95% of the transactions. The new and old comments tables thus contain about 31,500 and 475,000 comments, respectively. The total size of the database, including indices, is 1.4GB. More details about the database configuration can be found in [4].

4.2 Implementation Complexity

Table 1 summarizes the number of classes and the code size (including comments) of the presentation logic (the servlets) and the business logic (the EJBs) for each implementation.

Table 1. Number of classes and code size between presentation and business logic for each implementation

	Presentation logic		Business logic		Total	
	Classes	Lines of code	Classes	Lines of code	Classes	Lines of code
Servlets only	25	4590	-	-	25	4590
Session beans	24	2650	52	5270	76	7920
DAO (EB CMP)	24	4060	40	7260	64	11320
Session façade	24	2660	85	10780	109	13440

Each bean requires 3 classes: the home interface, the remote interface and the bean implementation. We also implement a primary key class for each entity bean. This makes the implementation of the business logic very verbose, reaching up to 80% of the total application code size.

Although EJBs are easy to write, the number of beans can become quite large, resulting in a larger code base and negatively affecting development time and maintenance cost. There are also portability problems between the two containers, which each have their own limitations and peculiarities, such as naming conventions. Even in the common part of the deployment descriptors, both containers have slightly different conventions, especially for inter-bean references.

5. EXPERIMENTAL ENVIRONNEMENT

5.1 Client Emulation

We implement a client-browser emulator. A session is a sequence of interactions for the same customer. For each customer session, the client emulator opens a persistent HTTP connection to the Web server and closes it at the end of the session. Each emulated client waits for a certain think time before initiating the next interaction. The next interaction is determined by a state transition matrix that specifies the probability to go from one interaction to another one.

The think time and session time for all benchmarks are generated from a negative exponential distribution with a mean of 7 seconds and 15 minutes, respectively. These numbers are based on clauses 5.3.1.1 and 6.2.1.2 of the TPC-W v1.65 specification [21]. We vary the load on the site by varying the number of clients. We have verified that in none of the experiments client emulation is the bottleneck.

5.2 Software Environment

We use Apache v.1.3.22 as the Web server. We increase the maximum number of Apache processes to 512. With that value, the number of Apache processes is never a limit on performance.

The servlet container is Jakarta Tomcat v3.2.4 [12], running on Sun JDK 1.3.1. The EJB servers are JOnAS v2.4.4 [15] and JBoss v2.4.4 [13]. These are are the latest stable versions of these products at the time of this writing. JOnAS v2.4.4 embeds Jonathan 3.0a5 that can be used for optimized communication. Both containers implement the EJB 1.1 specification and run on Sun JDK 1.3.1. For each implementation, we only start those container services that are necessary to perform the experiment. We avoid reloading the beans from the database if they were not modified (*tuned updates* in JBoss, *shared flag/isModified* in JOnAS). For all experiments, the transaction timeout is set to 5 minutes.

We use MySQL v.3.23.43-max [16] as our database server with the MM-MySQL v2.07 type 4 JDBC driver and MyISAM non-transactional tables. This means that transaction commands like begin/commit are accepted but have no effect, and a rollback generates an exception.. MySQL never becomes the bottleneck in our experiments

All machines run the 2.4.12 Linux kernel.

5.3 Hardware Platform

The Web server, the servlets server, the EJB server and the database server each run on a different machine, a PIII 933MHz CPU with 1GB SDRAM, and two Quantum Atlas 9GB 10,000rpm Ultra160 SCSI disk drives. A number of PII 450MHz machines run the client emulation software. We use enough client emulation machines to make sure that the clients do not become a bottleneck in any of our experiments. All machines are connected through a switched 100Mbps Ethernet LAN.

5.4 Measurement Methodology

We perform measurements for the three implementations of the application for each EJB container using non-optimized and optimized communication layers.

Each experiment is composed of 3 phases. A warm-up phase initializes the system until it reaches a steady-state throughput level. We then switch to the steady-state phase during which we perform all our measurements. Finally, a cool-down phase slows down the incoming request flow until the end of the experiment. For all experiments we use the same length of time for each phase. The auction site uses 2, 15 and 1 minutes. These lengths of time are chosen by observing when the experiment reaches a steady state and by observing the length of time necessary to obtain reproducible results.

To measure the load on each machine, we use the *Sysstat* utility [20] that collects CPU, memory, network and disk usage from the Linux kernel every second. The resulting data files are analyzed post-mortem to minimize system perturbation during the experiments.

We perform separate experiments to profile the containers using the OptimizeIt offline profiling. We use the instrumentation profiling, which is more suitable for profiling a large number of

threads and small functions than a sampler profiler. Due to the overhead of the profiler, the peak point is reached earlier for a given configuration. For each implementation, we choose the lowest number of clients for which we observe a peak point with any of the container configurations. For each configuration we analyze a snapshot of a 10-minute run with this number of clients.

6. EXPERIMENTAL RESULTS

We present the experimental results for each implementation in the same order they were introduced in section 3.1. For each implementation, we evaluate up to 5 different configurations referred to as follows:

- *Java Servlets*: the Java Servlets implementation,
- *JBoss*: the JBoss container using JNP and passing objects by value,
- *JOnAS-RMI*: the JOnAS container using RMI,
- *JBoss optimized calls*: the JBoss container using JNP and passing objects by reference,
- *JOnAS-Jeremie*: the JOnAS container using the Jeremie communication layer.

Each point in the graphs represents the best result of three runs of the experiment for the given number of clients and container configuration. The difference between runs is minor. A more complete report of all experimental results, including throughput, response time and resource usage are available from our Web site at http://www.cs.rice.edu/CS/Systems/DynaServer/.

6.1 Sessions beans with BMP

Figure 4 reports the throughput in interactions per minute as a function of number of clients for the browsing mix workload, for the 5 configurations previously introduced.

For both versions of JBoss, the peak point is reached at 800 clients with nearly 8600 interactions per minute. *JOnAS-RMI* peaks at about 8900 interactions per minute, for the same number of clients. *JOnAS-Jeremie* scales further, reaching 10150 interactions per minute with 1000 clients. The *Java Servlets* implementation shows even better performance with 12000 interactions per minute for 1200 clients.

At the peak point, the CPU utilization with JBoss is around 65% and the bottleneck appears on the servlet server. The high load on the servlet server is due to the JBoss stub used by the servlets to access the JBoss container. For *JOnAS-RMI*, the CPU on the EJB server is the bottleneck at the peak point, and the servlet server CPU utilization is 80%. *JOnAS-Jeremie* saturates both the EJB, the servlet and the database server processor at the peak point. The network bandwidth on the Web server is also very high with 80Mb/s exchanged with the clients and 14Mb/s with the servlets.

Though the bottlenecks are different, we do not observe a significant difference in performance between *JOnAS-RMI* and both versions of JBoss. Due to its more scalable communication layer *JOnAS-Jeremie* scales better and offer 33% more throughput after the peak point, compared to *JBoss optimized calls*. The *Java Servlets* version does not have the RMI overhead, and has direct access to the database without going through an EJB container. This explains the better performance of the servlets implementation.

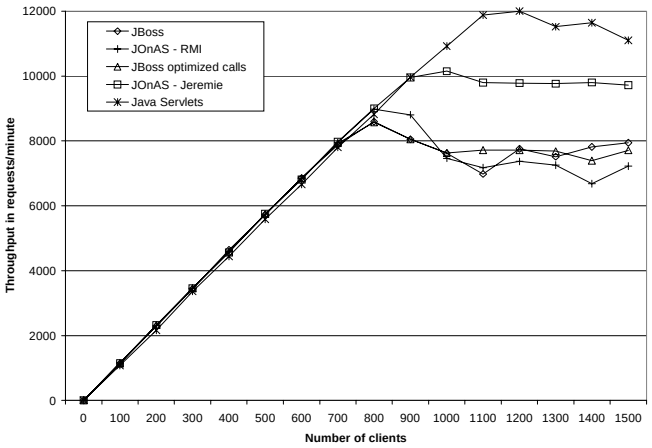

Figure 4. SB BMP implementation throughput in interactions per minute as a function of number of clients for the browsing mix using JBoss and JOnAS containers compared with a Java servlets implementation.

As shown in figure 5, the throughput for the bidding mix changes the ordering of the best performers. *JBoss* and *JOnAS-RMI* still have the lowest throughput at 6600 interactions per minute with 700 clients, *JBoss optimized calls* offers a significant improvement with a peak at 7500 interactions per minute with 800 clients.

JOnAS-Jeremie gives performance comparable to the *Java Servlets* until 1100 clients where it peaks at 9750 interactions per minute. *Java Servlets* reaches 10440 interactions per minute with 1200 clients.

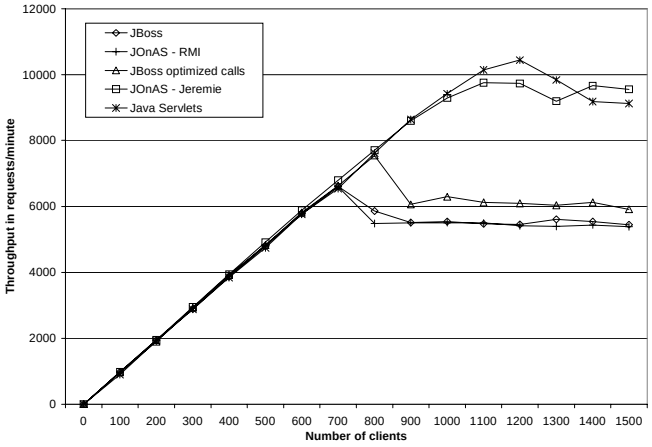

Figure 5. SB BMP implementation throughput in interactions per minute as a function of number of clients for the bidding mix using JBoss and JOnAS containers compared with a Java Servlets implementation.

Figure 6 shows the execution time breakdown resulting from profiling the SB BMP implementation for the bidding mix at 700 clients (the peak point of the *JBoss* and *JOnAS-RMI* configurations). As expected, communication costs dominate the execution time in this implementation where few services from the container are involved. It is also interesting to observe that the bean code we have written represents less than 1.5 percent of the total execution time.

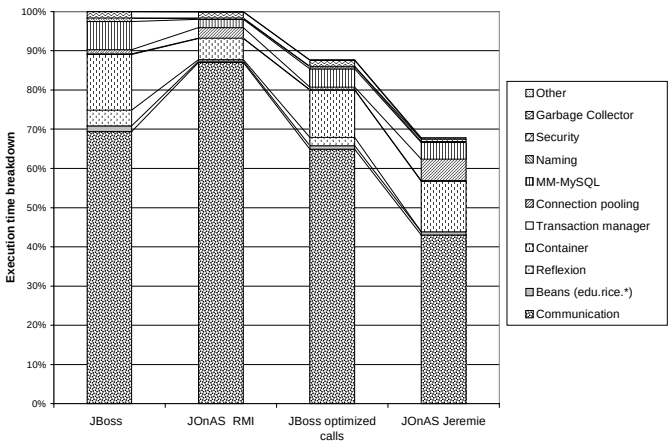

Figure 6. Execution time breakdown for the SB BMP implementation for the bidding mix at the peak point of the *JBoss* and *JOnAS-RMI* configurations.

JBoss spends 14.2% of overall CPU time in the container with an extra 4% for reflection. *JOnAS-RMI* shows a cost of 5% for the container and 2.7% for the database connection pooling. The communication layer is definitely the bottleneck with 69.4% of the overall processing time for *JBoss* and 87% for *JOnAS-RMI*. This difference in communication cost between *JBoss* and *JOnAS-RMI* is explained by the stub implementation on the client side. JBoss's stub can handle some calls locally, avoiding calls over the network [14]. The reduction in the time spent in communication is compensated by the overhead of the container. The throughput remains the same as *JOnAS-RMI* where the extra communication cost is compensated by a more efficient container.

JBoss optimized calls gets a 4.6% improvement in communication cost compared to *JBoss*. The container overhead remains proportionally the same (12.1% in the container, 2.1% in reflection). Due to its more scalable communication layer, *JOnAS-Jeremie* spends only 43% in communications, but the generated container classes are more expensive than the one generated for RMI. Therefore, the EJB container takes 12.9% of the overall execution time and 5.5% is consumed by the connection pooling.

In summary, with session beans and bean-managed persistence, the communication cost dominates the costs associated with the container. An efficient communication layer leads to better performance. The container design does not have a significant impact.

6.2 DAO separation with EB CMP

Figure 7 reports the throughput in interactions per minute as a function of number of clients for the browsing mix workload, using the 4 container configurations described in section 6. The absolute throughput numbers are between 8 (for *JOnAS-Jeremie*) and 16 times (for *JBoss*) lower than with the previous SB BMP implementation. We terminate the experiments for *JBoss* and *JOnAS-RMI* after 200 clients, because *JBoss* becomes unable to handle the load and transactions abort on timeout (the timeout is set to 5 minutes).

Both JBoss configurations give comparable peak performance, with 547 and 579 interactions per minute reached with 80 clients for *JBoss* and *JBoss optimized calls*, respectively. *JOnAS-RMI* peaks at 939 interactions per minute with 100 clients. Best results are achieved by *JOnAS-Jeremie* at 1222 interactions per minute for 200 clients.

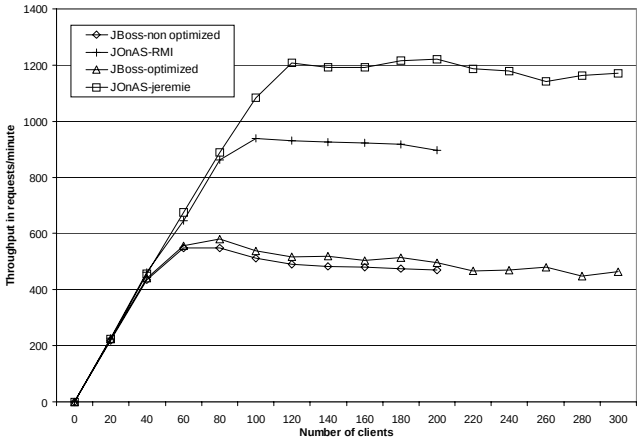

Figure 7. DAO separation with EB BMP implementation throughput in interactions per minute as a function of number of clients for the browsing mix using JBoss and JOnAS.

Figure 8 shows the throughput in interactions per minute as a function of number of clients for the bidding mix. The ordering of the different configuration in terms of performance is the same as for the browsing mix. *JBoss* reaches a peak of 638 interactions per minute for 100 clients. There is a 37% improvement when passing objects by reference, resulting in 868 interactions per minute for *JBoss optimized calls* with 120 clients. The improvement is due to the fact that for each write interaction, there is a call to a bean assigning unique identifiers that can be optimized. This interaction does not occur in the browsing mix, and therefore there is no comparable improvement. *JOnAS-RMI* achieves 1130 interactions per minute for 140 clients, and *JOnAS-Jeremie* achieves 1401 interactions per minute with 180 clients.

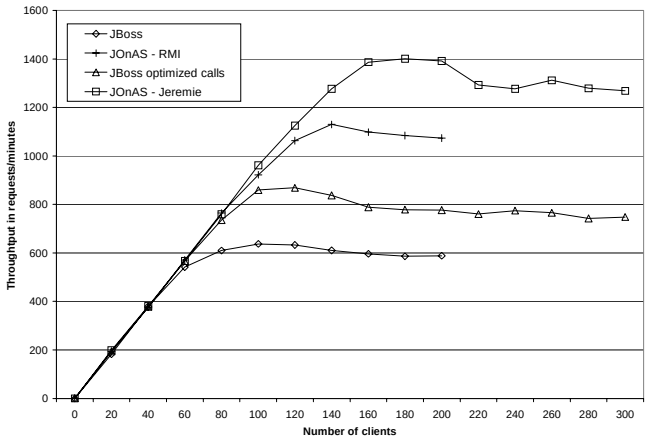

Figure 8. DAO separation EB BMP throughput in interactions per minute as a function of number of clients for the bidding mix using JBoss and JOnAS.

Figure 9 shows the execution time breakdown for the EB CMP implementation for the bidding mix at 80 clients (the peak point of the *JBoss* configuration). Compared to the SB BMP implementation, the container is more heavily involved in the processing due to the persistence management. The time spent in the communication layers is significantly reduced. As most of the

code is generated by the container, the overall execution time spent in our bean classes is less than 0.1%.

The time spent in the JBoss container is more than 40% of which one fourth is due to reflection. In comparison, the JOnAS container takes less than 11% of the processing time (and 0.3% in reflection). Of that 11%, 3.8% is due to the connection pooling and 1% to the transaction manager. The JBoss's client stub optimization does not seem to work with entity beans and the respective communication time are 54% for *JBoss* and 47.4% for *JOnAS-RMI*.

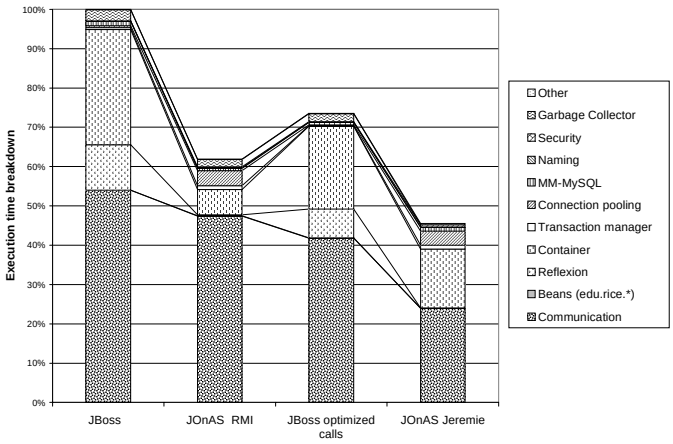

Figure 9. Execution time breakdown for the entity beans implementation for the bidding mix at the peak point of the *JBoss* configuration.

JBoss optimized calls reduces the time spent in communication to 41.8% of the overall execution time. However, 21.1% is spent in the container and an additional 7.4% in reflection. Performance is improved compared to *JBoss*, but still below what is obtained with *JOnAS*, even without communication optimizations. *JOnAS-Jeremie* shows a significant improvement with only 23.9% of the CPU used for communication and 15% for the container. Jeremie offers again more scalable performance than RMI. Like for the SB BMP implementation, the container classes for Jeremie are more expensive than the container classes for RMI, but the container/communication combination is more efficient and results in better performance.

In summary, unlike for the SB BMP version, the container design has the largest impact on performance for EB CMP. Optimized communications still improve performance but to a lesser extent.

6.3 Session façade implementation

Figure 10 presents the throughput in interactions per minute as a function of number of clients for the browsing mix using the 4 containers configuration described in section 6.

Due to the communication overhead between the façade session beans and the entity beans, both *JBoss* and *JOnAS-RMI* perform worse than with the EB CMP implementation. *JBoss* peaks at 378 interactions per minute with 60 clients, while *JOnAS-RMI* achieves 689 interactions per minute with 100 clients. This represents almost a 30% drop in performance for both configurations. As for the previous experiment we do not report throughput for more than 200 clients due to transaction timeouts.

JBoss optimized calls shows a significant improvement with a peak at 1081 interactions per minute with 120 clients. The optimization improves the throughput by a factor of 2.86 compared to JBoss without optimized calls. *JOnAS-Jeremie* peaks at 3970 interactions per minute with 440 clients providing a speedup of 5.3 compared to *JBoss optimized calls*. The ability of Jeremie to optimize the local calls clearly shows its benefits here.

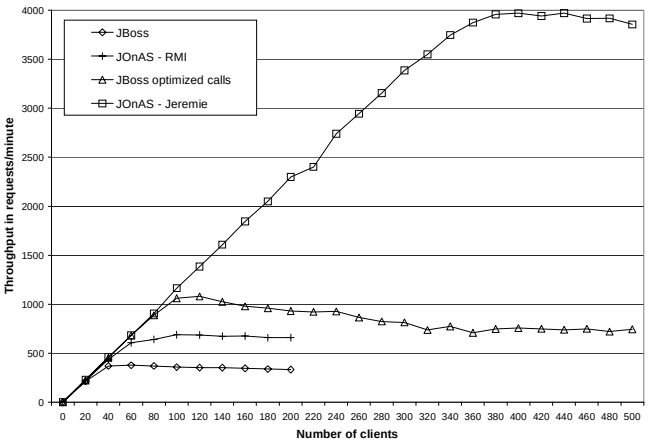

Figure 10. Session façade implementation throughput in interactions per minute as a function of number of clients for the browsing mix using JBoss and JOnAS.

Figure 11 reports the throughput in interactions per minute as a function of number of clients for the bidding mix. The scenario is the same for *JBoss* and *JOnAS-RMI*. They peak at 448 and 777 interactions per minute, with 60 and 140 clients respectively. Inter-bean communication adds to the overall communication overhead and pulls performance down giving the worst throughput of all implementations.

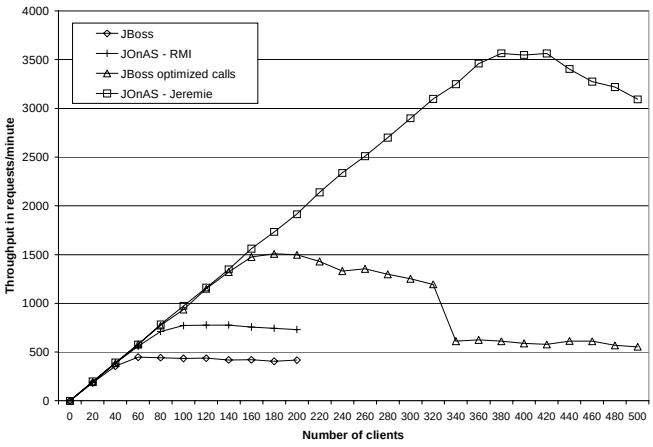

Figure 11. Session façade implementation throughput in interactions per minute as a function of number of clients for the bidding mix using JBoss and JOnAS.

JBoss optimized calls offers a better performance with a peak at 1507 interactions per minute with 180 clients. However, response time dramatically increases under saturation till 340 clients where the transactions take more than 5 minutes to be performed and are timed out by the transaction manager. Then, the number of completed interactions drops around 600 per minute.

JOnAS-Jeremie has more scalable behavior and sustains up to 3565 interactions per minute between 380 and 420 clients. This leads up to a 6.2 factor of improvement compared to *JBoss optimized calls* with the same number of clients.

Figure 12 presents the execution time breakdown for the session façade implementation for the bidding mix at 60 clients (the peak throughput of the *JBoss* configuration). Once again our bean code represents less than 1% of the overall execution time.

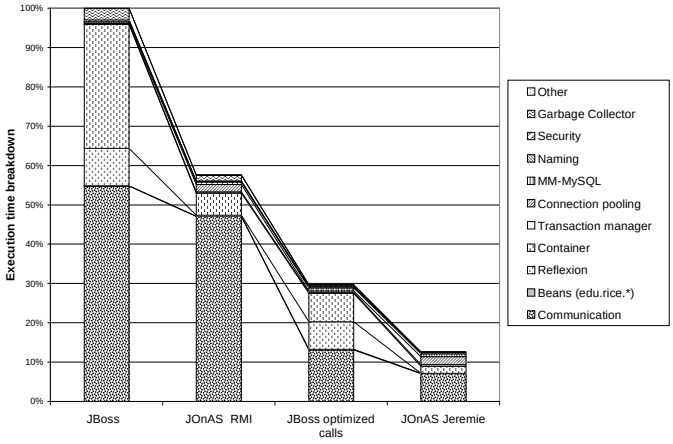

Figure 12. Execution time breakdown for the session façade implementation for the bidding mix at the peak point of the *JBoss* configuration.

There is little difference with the EB BMP implementation for both JBoss and JOnAS-RMI. As communications were already the bottleneck, they increase by less than 1% and remain the bottleneck.

As the number of beans and interactions between beans increase, the time spent in reflection with the *JBoss optimized calls* configuration reaches 7% of the overall execution time. The container itself takes 7.3% of the total time. The call optimization is visible in the reduction of the CPU utilization dedicated to communication that drops to 13.1%. For the first time, we observe than more time is spent in the container than in communications.

JOnAS-Jeremie cuts the communication time by almost a half to only 7%. The container CPU utilization then represents only 1.8% of total execution time whereas connection pooling takes 2%. The larger number of lookups on beans affects 0.35% of the overall time to the naming directory.

In summary, with session façade beans both the container and communication layer designs have a significant impact on performance. With a larger number of beans, reflection proves to be a significant limitation to scalability. The pre-compiled approach reduces the time spent in reflection and offers scalable performance when coupled with an optimized communication layer such as the one implemented in Jeremie.

6.4 Summary

Figure 13 and figure 14 summarize the peak throughput obtained for the different implementations and container configurations for the browsing mix and the bidding mix respectively.

The session beans with bean-managed persistence (SB BMP) implementation gives the best throughput. The communication layer is the bottleneck and hides most of the cost of the container. Therefore, container design has little impact on performance for this implementation.

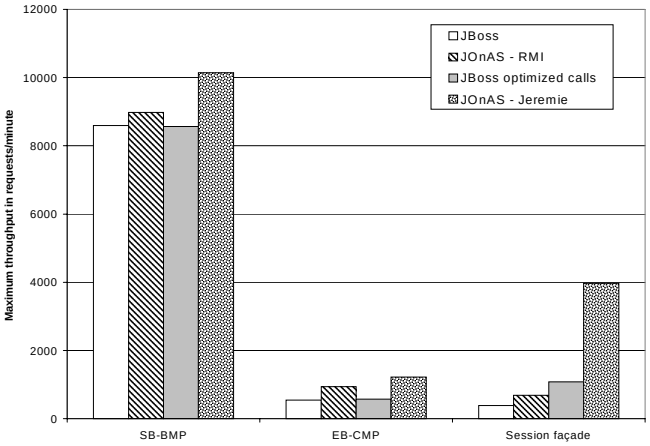

Figure 13. EJB implementations maximum throughput in interactions per minute for the browsing mix.

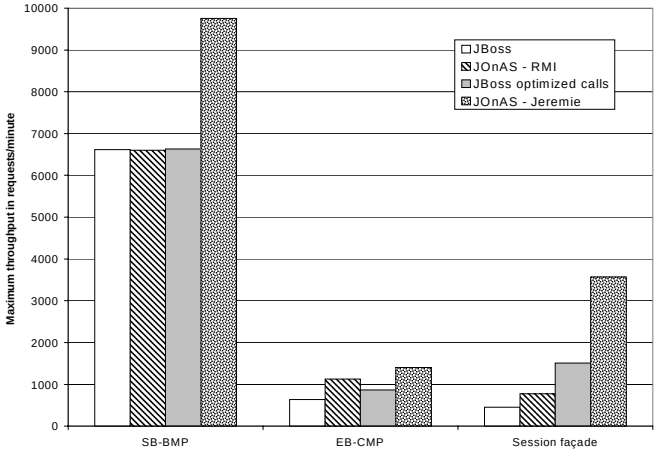

Figure 14. EJB implementations maximum throughput in interactions per minute for the bidding mix.

The DAO separation with entity beans and container-managed persistence (EB CMP) implementation gives the least scalable results. This is due to the excessively fine-grain access exposed by the entity beans to the servlets. This implementation, however, shows that container design has a significant impact on the performance of EB CMP. The pre-compiled approach of JOnAS shows better scalability than the dynamic proxy based approach used by JBoss.

The overhead of reflection is also noticeable in the session façade implementation. The optimized calls improve throughput for JBoss at lower loads, but performance does not scale and response time quickly rises after the peak point. The call optimization is not sufficient to mask the overhead of reflection in the container. Only the combination of pre-compiled container classes and an optimized communication layer such as Jeremie allows for scalable performance.

The bean code written by the programmer represents at most two percent of the overall execution time. This confirms that

application implementation method and the middleware design have the biggest impact on performances. The two have to be considered in combination, as evidenced by the poor performance of session façade beans without optimized inter-bean communication.

7. RELATED WORK

Performance and scalability of J2EE application servers is a very hot topic in the e-business community. Sun has released the ECperf specification [18] as a first attempt to standardize the evaluation of EJB servers. This benchmark is aimed at evaluating a particular J2EE application server with a single application, while we target the evaluation of different EJB containers with various implementations of the same application. Other benchmarks such as TPCW [21] overload the database tier [4] preventing evaluation of middle-tier performance under saturation.

To the best of our knowledge, ours is the first study of EJB applications scalability analyzing the container and communication layers designs. [9] gives guidelines for EJB server comparison, but they use the EJB 1.0 specification and do not propose an application to perform the comparison. We have made available the application, container configurations and experiment results on our Web site http://www.cs.rice.edu/CS/Systems/DynaServer to allow further evaluation of other containers.

UrbanCode provides a performance benchmark of design idioms (design patterns applied to a specific programming language) [22]. Their conclusions about relative performance between session beans and entity beans confirm our results. They do not, however, evaluate the impact of container design or the communication layer optimization. Allamaraju et al. [2] discuss container design but conclude that reflection is never an issue, because its cost is insignificant compared to network latency or roundtrips to the database. We have shown that reflection can become a real issue for scalability when container-managed persistence is used.

To address the issue of inter-bean communications, the EJB CMP 2.0 specification [19] introduces the notion of local and remote interfaces. As this feature was not implemented in the containers we have tested, this evaluation will be part of our future work when the implementation becomes available.

Another solution to achieve scalability is to use a cluster. Major J2EE vendors implement such as BEA [6] or IBM [8] use clustering to achieve scalability and high availability. We also plan to evaluate those features, but they are still under development in the open source containers we used for this evaluation.

8. CONCLUSIONS

We have experimented with several implementations of the same e-commerce application, using different application implementation methods, container designs and communication layers. The source code, container configurations, database contents and experiment reports are all available for download from our Web site at http://www.cs.rice.edu/CS/Systems/DynaServer/.

We have shown that stateless session beans with bean-managed persistence coupled with an efficient communication layer offer performance comparable to Java Servlets. Entity beans impose a row-level access to the database resulting in a finer grain access and significantly lower performance.

Container design has no significant influence on SB BMP, because communication costs dominate, but we have shown that it has a direct impact on performance with entity beans. The dynamic proxy approach has a large overhead that limits scalability. Pre-compiled approaches reduce the use of reflection at run time, thus providing better scalability.

Communication layers are the determining factor for the scalability of the session façade implementation. Only the container with pre-compiled classes combined with an optimized communication layer offer scalable performance. Reflection cost increases with the number of beans, quickly resulting in a bottleneck.

9. ACKNOWLEDGMENTS

We are grateful for the people of the JBoss and JOnAS communities for the useful help and information they provided us.

10. REFERENCES

[1] Rahim Adatia et al. – Professionnal EJB – *Wrox Press, ISBN 1-861005-08-3*, 2001.

[2] Subrahmanyam Allamaraju et al. – Professional Java Server Programming J2EE Edition - *Wrox Press, ISBN 1-861004-65-6*, 2000.

[3] Deepak Alur, John Crupi and Dan Malks – Core J2EE Patterns – *Sun Microsystem Press, ISBN 0-13-064884-1*, 2001.

[4] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan L. Cox, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani and Willy Zwaenepoel – Bottleneck Characterization of Dynamic Web Server Benchmarks – *Technical Report TR02-388*, Rice University, 2001.

[5] The Apache Software Foundation – http://www.apache.org/.

[6] BEA Systems, Inc – Achieving Scalability and High Availability for E-Business – BEA white paper, http://www.bea.com, 2001.

[7] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite and Willy Zwaenepoel – A Comparison of Software Architecture for E-business Applications – *Technical Report TR02-389*, Rice University, 2001.

[8] Willy Chiu – Design for Scalability – IBM white paper, http://ibm.com/websphere/developer/zones/hvws, 2001.

[9] Distributed Systems Research Group, Charles University – EJB Comparison Project – http://nenya.ms.mff.cuni.cz, 2000.

[10] Bruno Dumant, François Horn, Frédéric Dang Tran and Jean-Bernard Stefani – Jonathan : an Open Distributed Processing Environment in Java – *Distributed Systems Engineering Journal*, vol. 6, 3-12, 1999.

[11] eBay – http://www.ebay.com/.

[12] Jakarta Tomcat servlet container – http://jakarta.apache.org/tomcat/.

[13] JBoss EJB server – http://jboss.org.

[14] Vladimir Blagojevic and Rickard Oberg – Container architecture - design notes – http://www.jboss.org/online-manual/HTML/ch12.html.

[15] JOnAS: Java Open Application Server – http://www.objectweb.org/jonas.

[16] MySQL Reference Manual v3.23.36 – http://www.mysql.com/documentation/.

[17] Sun Microsystems – Dynamic Proxy Classes – http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html, 2001.

[18] Sun Microsystems – ECperf specification – http://java.sun.com/j2ee/ecperf/, 2001.

[19] Sun Microsystems – EJB 2.0 specification – http://java.sun.com/products/ejb/docs.html, 2001.

[20] Sysstat package – http://freshmeat.net/projects/sysstat/.

[21] Transaction Processing Performance Council – http://www.tpc.org/.

[22] UrbanCode, Inc. – EJB Benchmark – http://www.urbancode.com/projects/ejbbenchmark, 2001.