Performance and Scalability of EJB Applications

Emmanuel Cecchet
Rice University/INRIA
655, avenue de l'Europe
38330 Montbonnot, France
emmanuel.cecchet@inrialpes.fr

Julie Marguerite
Rice University
6100 Main Street, MS-132
Houston, TX, 77005, USA
margueri@rice.edu

Willy Zwaenepoel
Rice University
6100 Main Street, MS-132
Houston, TX, 77005, USA
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 five versions of the auction site. The first version uses stateless session beans, making only minimal use of the services provided by the Enterprise JavaBeans (EJB) container. Two versions use entity beans, one with container-managed persistence and the other with bean-managed persistence. The fourth version applies the session façade pattern, using session beans as a façade to access entity beans. The last version uses EJB 2.0 local interfaces with the session façade pattern. 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-compiles 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 application implementation method. EJB applications with session beans perform as well as a Java servlets-only implementation and an order-of-magnitude better than most of the implementations based on entity beans. The fine-granularity access exposed by the entity beans limits scalability. Use of session façade beans improves performance for entity beans, but *only* if local communication is very efficient or EJB 2.0 local interfaces are used. 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. With entity beans, the design of the container becomes important. In particular, the cost of reflection affects performance. For implementations using session façade beans, local communication cost is critically important. EJB 2.0 local interfaces improve the performance by avoiding the communication layers for local communications.

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 and above all scalable platforms to host these sites. The Java™ 2 Platform Enterprise Edition (J2EE) specification addresses these issues. J2EE primarily targets n-tier application development [2]. It defines a set of Java APIs to build applications and provides a run-time infrastructure for hosting these applications.

The J2EE run-time environment includes four different containers: the application client container, the applet container, the Web container and the Enterprise JavaBeans (EJB) container (see Figure 1). The EJB server is often the bottleneck in J2EE applications [9]. 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 five different EJB implementations of an auction site modeled after eBay [13]. The semantics are the same for each implementation. The five different application implementation methods are: stateless session beans, entity beans with container-managed persistence, entity beans with bean-managed persistence, entity beans with session façade beans, and EJB 2.0 local interfaces (entity beans with only local interfaces and session façade beans with remote interfaces). For further comparison, we have also implemented a Java servlets-only version that does not use EJB.

We evaluate two different container designs that are representative of most approaches used in EJB containers available at this time. The dynamic proxy approach [19], used in the popular JBoss [15] open-source EJB server, generates the container classes at run time, making extensive use of reflection. Most commercial implementations and the JOnAS [16] open-source EJB container use pre-compilation: classes are generated during deployment, reducing the use of reflection at run time. We also configure the EJB servers with and without communication optimizations.

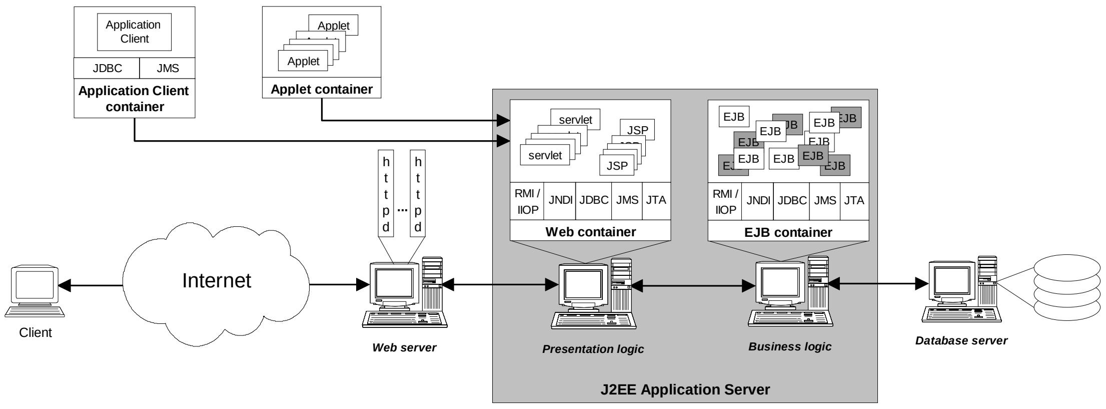

Figure 1. Enterprise Java Beans in the J2EE framework.

We use open-source software in common use for our experiments: the Apache Web server [5], the Tomcat servlet server [14], the JBoss [15] and JOnAS [16] EJB servers and the MySQL [18] relational database. We have posted all software, configuration files, and full experimental 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 CPU on the EJB server is the bottleneck. The memory and disk are never a limiting resource. The network can reach very high utilization (up to 94%) when few services from the EJB container are used.

The most important factor in determining performance is the application implementation method. The implementation using session beans performs as well as a Java servlets-only implementation and an order-of-magnitude better than most of the implementations based on entity beans. Use of session façade beans improves performance, but only if local communication is very efficient or EJB 2.0 local interfaces are used.

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. With entity beans, the design of the container becomes important. In particular, the cost of reflection affects performance. For implementations using session façade beans, local communication cost is critically important. JDK 1.4 reduces reflection cost but the overall improvement remains modest.

The outline of the rest of this paper is as follows. Section 2 provides some background on EJB. Section 3 provides a detailed description of the alternative application 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 section 7. 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 either 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 in 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 Application implementation method

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

3.1.1 Session beans

We use session beans to implement the business logic, leaving only the presentation logic in the servlets as depicted in figure 2.

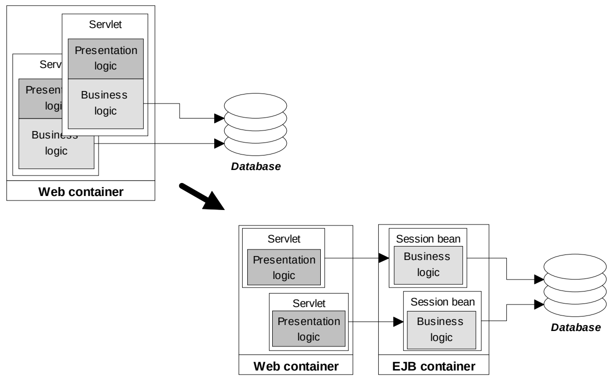

Figure 2. Servlets-only to session bean implementation.

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 servlets-only code, in which the connection pooling has to be implemented by hand.

3.1.2 *DAO separation with Entity Beans CMP*

In this implementation, we extract the data access code from the servlets, and move it into Data Access Objects (DAO) [25] that we implement using entity beans. The business logic embedded in the servlets directly invokes methods on the entity beans that map the data stored in the database. Figure 3 illustrates the DAO separation with entity beans.

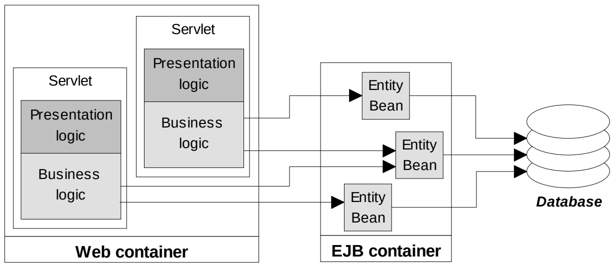

Figure 3. DAO separation with Entity Beans.

With container-managed persistence (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. With this implementation, we evaluate the impact of fine-grain accesses between the Web and EJB containers.

3.1.3 *DAO separation with Entity Beans BMP*

This implementation is the same as the DAO separation with entity beans CMP version except that we use bean-managed persistence (BMP). With BMP, the SQL queries have to be hand-coded in the beans. We implement exactly the same queries as the CMP version including the use of a stateless bean to execute complex queries. The goal of this implementation is to evaluate the cost of the container's persistence service by comparing it with the entity beans CMP version.

3.1.4 *Session façade*

The session façade pattern [3] uses stateless session beans as a façade to abstract the entity components as shown in figure 4.

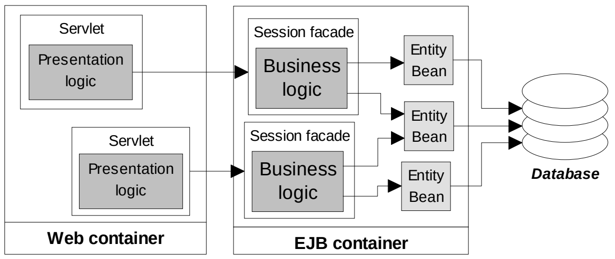

Figure 4. Session façade design pattern.

This method 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). We use container-managed persistence for the entity beans.[1]

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.

3.1.5 *EJB 2.0 local interfaces*

Although the session façade beans and the entity beans execute inside the same JVM, with RMI (Remote Method Invocation) communication between them has to go through all the communication layers, as if they were on different machines. The EJB 2.0 specification [22] introduces *local interfaces* to optimize intra-JVM calls by bypassing the communication layers. Beans with a local interface cannot be called remotely, i.e., from another JVM even if the JVM runs on the same machine.

Our final implementation takes advantage of these local interfaces. This implementation uses the session façade pattern, and container-managed persistence. Only session façade beans have a remote interface that is exposed to the servlets. The entity beans only have a local interface that is used by the session façade beans. Therefore, interactions between session and entity beans bypass the communication layers. This implementation is the only one that requires EJB 2.0 compliant containers.

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 then the container forwards 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 pre-compiled 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 the JOnAS EJB container and to the best of our knowledge in most commercial EJB containers.

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 to generate the home and component interfaces at run time. A dynamic proxy is an object generated at run time that implements

[1] We expect the results with bean-managed persistence to be similar.

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 2.4 container, which supports only EJB 1.1, 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, or stateful session bean).

The new JBoss 3.0 container, which supports EJB 2.0, uses the Byte Code Engineering Library (BCEL) [6] to generate specialized dynamic proxies for each bean. The goal of this approach is to reduce the use of reflection at run time.

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 [12]. 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 with 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 similar to eBay.

4.1 Description

Our auction site defines 27 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 [13]). Browsing items also includes consulting the bid history and the seller's information. 5 of the 27 interactions are implemented using static HTML pages. The remaining 22 interactions require data to be generated dynamically. 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 size 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, resulting in 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 presents the total number of classes and the total code size (including comments) for each implementation, and the breakdown of the number of classes and the code size between servlets and beans.

Table 1. Number of classes and code size of servlets and beans for each application implementation method.

	Servlets		Beans		Total	
	Classes	Lines of code	Classes	Lines of code	Classes	Lines of code
Servlets-only	25	4590	-	-	25	4590
Session beans	22	2730	51	5270	76	8000
EB CMP	23	3980	40	6780	63	10760
EB BMP	23	3980	40	9850	63	13830
Session façade	22	2660	85	10780	107	13440
EJB 2.0 local	22	2725	91	11070	113	13795

4.2.1 Servlets

The number of servlet classes varies little between the various application implementation methods, from 22 to 25. The number of lines of code in the different versions, however, varies considerably.

In all versions, there is a one-to-one match between the dynamic interactions and the servlets except for the BrowseCategories servlet that implements 3 interactions (browse categories, browse categories in region, and browse categories to sell item). Therefore, the 22 dynamic interactions are implemented with 20 servlets. All versions have two extra classes: one that manages HTML output and another one that handles platform-specific configuration variables.

Both the EB and the servlets-only versions use a servlet for user authentication. In the EB versions, only the data access code is moved to the entity beans; the business logic is still performed by the servlets. In all other implementations, the business logic is moved from the servlets to the session beans so user authentication is done in the beans. The servlets-only version also uses two additional classes for connection pooling and time management, which are taken care of by the container in the other versions.

The servlets-only version has the largest number of lines of code in the servlets. This is not surprising, since the servlets encode both the presentation logic and the business logic. The number of lines of code for the EB versions is quite high as well since only the data access code has been moved to the beans, and the presentation and the business logic remains in the servlets.

4.2.2 Beans

The session beans implementation contains 51 classes, but is the smallest version in terms of code size with 5270 lines. The number of bean classes is the same for both EB versions with 40 classes, but the number of lines of code varies considerably between the two implementations: 6780 lines for the CMP version and 9850 lines for the BMP version. The session façade version is a little smaller than the EJB 2.0 local interfaces implementation with 85 classes and 10780 lines of code, compared to 91 classes and 11070 lines of code.

Each bean requires 3 classes: the home (or local home) interface, the remote (or local) interface and the bean implementation. We also implement a primary key class for each entity bean. Having 3 or 4 classes for each bean makes the implementation of the business logic very verbose, reaching up to 80% of the total application code size.

The session beans version contains 17 bean implementation classes, each with their home and remote interfaces, for a total of 51 bean classes. There is a one-to-one match between the servlet and the bean classes, except for the user authentication bean that is called by 3 different servlets (PutBidAuth, BuyNowAuth, and PutCommentAuth) for different interactions. The two remaining servlets are the ones discussed in section 4.2.1 (HTML output management and platform-specific configuration variables).

Both EB versions contain 10 bean implementation classes. Each has a primary key class, a home and a remote interface, for a total of 40 bean classes. The EB BMP version requires more lines of code than the CMP version since the finder methods and all the SQL queries are no longer generated by the container but have to be written in the beans. The entity beans provide a large number of getter/setter methods and need to implement a larger interface than the session beans (methods such as ejb_create(), ejb_remove(), etc). Therefore, the code size for the EB versions is larger than for the session beans implementation.

Both the session façade and the EJB 2.0 local interfaces implementations contain the same entity beans as the EB CMP version. When we add the session façade bean code to the EB code, it results in the largest implementations in terms of lines of code along with the EB BMP version. The session façade version contains 15 façade session beans, each with three classes. Together with the 40 entity bean classes, this accounts for a total of 85 classes.

In the session façade implementation, two entity beans are accessed directly from the servlets to perform inserts (create new entity beans). Remote entity bean access is not allowed in the EJB 2.0 local interfaces implementation, since all entity beans have only a local interface. Therefore, we have introduced two more session façade beans (6 classes) in the EJB 2.0 local interfaces implementation to act as proxies for those entity beans. Those 17 session façade beans correspond to the beans of the session beans version.

4.2.3 Summary

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 ENVIRONMENT

5.1 Client emulation

We implement a client-browser emulator as follows. 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 the session time are generated from a negative exponential distribution with a mean of 7 seconds and 15 minutes, respectively. These values are chosen in analogy with the values specified for the think time and the session time in the TPC-W benchmark, which models an online bookstore (see clauses 5.3.1.1 and 6.2.1.2 of the TPC-W v1.65 specification [25]). 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 [14], running on Sun JDK 1.3.1. For all experiments, except the ones with EJB 2.0 local interfaces, we use the JOnAS v2.4.4 [16] and JBoss v2.4.4 [15] containers. JOnAS v2.4.4 embeds Jonathan 3.0a5 that can be used for optimized communication. Both containers implement the EJB 1.1 specification. JBoss 3.0 and JOnAS 2.5 are used for the EJB 2.0 local interfaces implementation.

For all experiments, except for the ones in section 6.5, we use the Sun JVM from JDK 1.3.1 for Linux with the following options:

- `-server`: use the server JVM instead of the client JVM.
- `-Xms128m`: set the initial Java heap size to 128 MB to prevent spending time in increasing heap size during application warm-up.
- `-Xmx768m`: set the maximum Java heap size to 768 MB instead of the default of 64 MB to avoid that the EJB containers run out of memory.
- `-Xss32k`: set the thread stack size to 32 KB instead of the default Linux thread stack size of 2 MB to avoid running out of process virtual address space and not being able to create new threads. Our application does not do any recursion. Therefore, a 32 KB stack size is sufficient.

For the experiments in section 6.5, we use JDK 1.4.0_01. This JDK requires a 96 KB minimum stack size, therefore we use the `–Xss96k` option for these experiments.

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 are 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 [18] as our database server with the MM-MySQL v2.0.12 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 servlet 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 five implementations of the application for each EJB container using both non-optimized and optimized communication layers. The only exception is the EJB 2.0 local interfaces implementation using JBoss, where we are not able to disable the communication optimization.

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, namely 2, 15 and 1 minute, respectively. 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 [24] that every second collects CPU, memory, network and disk usage from the Linux kernel. The resulting data files are analyzed post-mortem to minimize system perturbation during the experiments.

We perform a separate set of experiments to profile the containers using the OptimizeIt [19] offline profiling tool. We use instrumentation profiling, which is more suitable than a sampling profiler for applications with a large number of threads and many small functions. Due to the overhead of the profiler, the peak point is reached earlier for a given configuration. For each application implementation method, 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.

Each point in the graphs in section 6 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 (CPU, memory, processes, network, disk), is available from our Web site at http://www.cs.rice.edu/CS/Systems/DynaServer/RUBiS/Results.

6. EXPERIMENTAL RESULTS

We compare the results for the different implementation methods in section 6.1. Next, in sections 6.2 to 6.5, we present the results for each application implementation method, in the same order as they are introduced in section 3.1. For each implementation, we evaluate four different configurations referred to as follows:

- *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.

We report additional results comparing JDK 1.3 to JDK 1.4 in section 6.6. We summarize the results of the performance evaluation in section 6.7.

6.1 Overall results

Figure 5 presents the results for the different application implementation methods, using for each method the configuration that results in the best performance results. In addition, we compare the results to a Java servlets-only implementation.

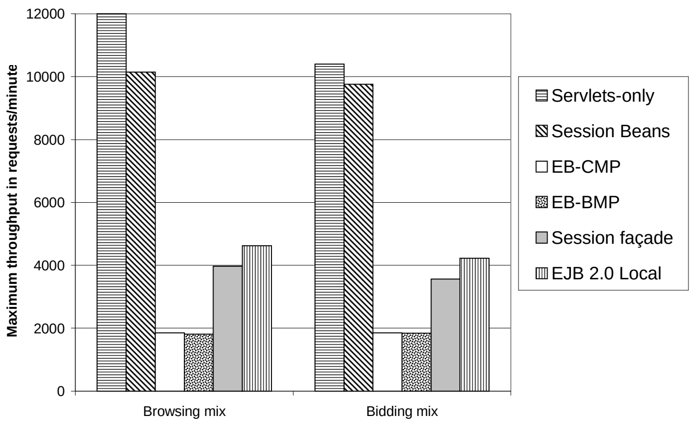

Figure 5. Maximum achievable throughput for each implementation method.

Session beans offer performance comparable to the Java servlets-only implementation. All other EJB-based implementation methods fare considerably worse. Session façade beans and EJB 2.0 local interfaces are more than a factor of 2 slower than the session beans implementations. Implementations based solely on entity beans experience an even bigger performance drop, regardless of whether they use container-managed or bean-managed persistence.

We now turn to a detailed analysis of the results for each application implementation method.

6.2 Session beans

Figure 6 reports the throughput in interactions per minute as a function of number of clients for the browsing mix workload, for the four configurations previously introduced and for the Java servlets-only implementation.

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 *Servlets-only* 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 the EJB, the servlet and the database server CPU 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 offers 33% more throughput after the peak point, compared to *JBoss optimized calls*. The *Servlets-only* 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-only implementation.

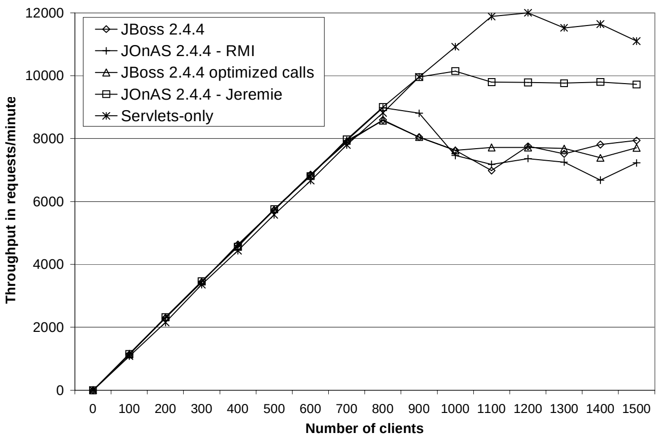

Figure 6. Session beans implementation throughput in interactions per minute as a function of number of clients for the browsing mix using JBoss and JOnAS containers compared with a servlets-only implementation.

As shown in figure 7, 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 *Servlets-only* version until 1100 clients where it peaks at 9750 interactions per minute. *Servlets-only* reaches 10440 interactions per minute with 1200 clients.

Figure 8 shows the execution time breakdown resulting from profiling the session bean implementation for the bidding mix at 700 clients (the peak point of the *JBoss* and *JOnAS-RMI* configurations). In this figure and in all further figures showing breakdowns of execution times, the results are normalized to the execution time of the slowest configuration for the application implementation method being discussed.

As expected, communication costs dominate the execution time in this implementation where few of the container's services are used. It is also interesting to observe that the bean code we have written represents less than 1.5 percent of the total execution time.

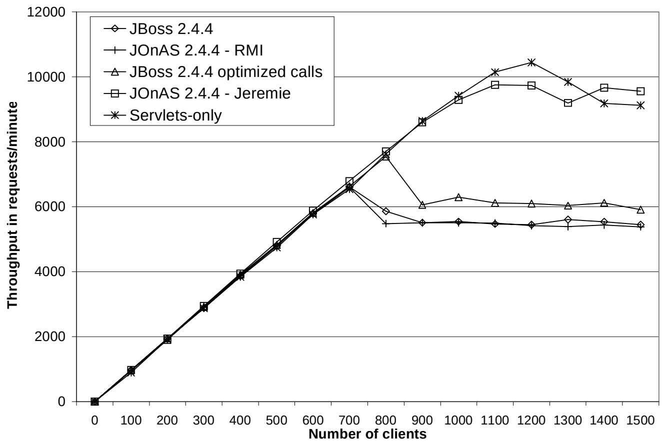

Figure 7. Session beans implementation throughput in interactions per minute as a function of number of clients for the bidding mix using JBoss and JOnAS containers compared with a servlets-only implementation.

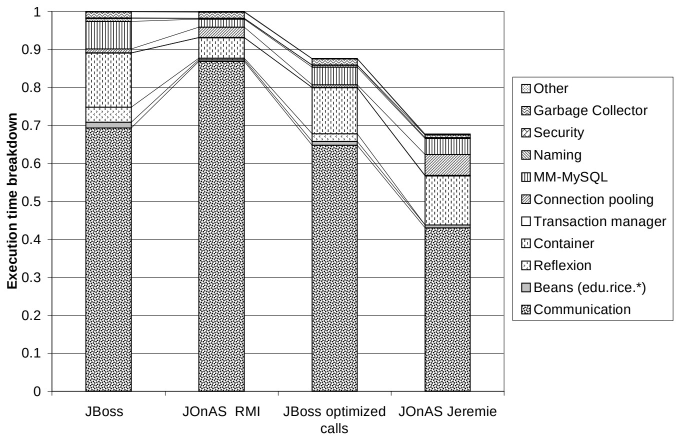

Figure 8. Execution time breakdown for the session beans implementation for the bidding mix at the peak point of the *JBoss* and *JOnAS-RMI* configurations.

JOnAS-RMI spends more time in communications than JBoss. This difference in communication cost is explained by the stub implementation on the client side. *JBoss*'s stubs can handle some calls locally, avoiding calls over the network [8]. The amount of time spent in the container is considerably smaller in *JOnAS* than in *JBoss*. In terms of overall performance, the differences in communication and container overhead cancel each other out, and the resulting throughput is the same for both configurations.

JBoss optimized calls shows a small improvement in communication cost compared to *JBoss*. The container overhead remains proportionally the same, leading to a small overall performance improvement. *JOnAS-Jeremie* spends considerably less time in communications, but the generated container classes are more expensive than the ones generated for RMI. Overall, though, performance is superior to the other configurations and approaches that of Java servlets-only.

All RMI-based configurations spend 79% of the communication time in the TCP/IP layers (java.net.* classes) and 21% in the RMI protocol and in serialization. *JOnAS-Jeremie* has a different distribution of communication costs, with 43.2% in TCP/IP and 56.8% in Jeremie and its serialization mechanism. Even though Jeremie's protocol is more expensive than JRMP (used in RMI), Jeremie greatly reduces the amount of information sent over the network. As a result, Jeremie cuts overall communication time by 35 to 51% compared to RMI-based configurations.

In summary, with session beans, 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.

6.3 Entity beans with CMP and BMP

Figure 9 reports the throughput using entity beans with CMP in interactions per minute as a function of number of clients for the browsing mix workload. Figure 10 reports the same results using entity beans with BMP. The absolute throughput numbers are between 5.5 (for *JOnAS-Jeremie*) and 16 times (for *JBoss*) lower than with the previous session beans implementation.

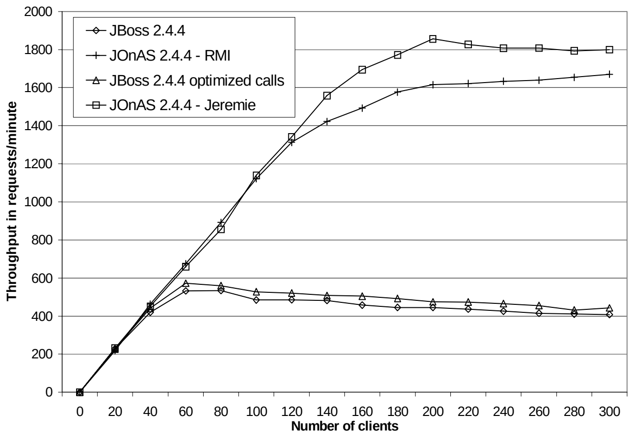

Figure 9. DAO separation with EB CMP implementation throughput in interactions per minute as a function of number of clients for the browsing mix using JBoss and JOnAS.

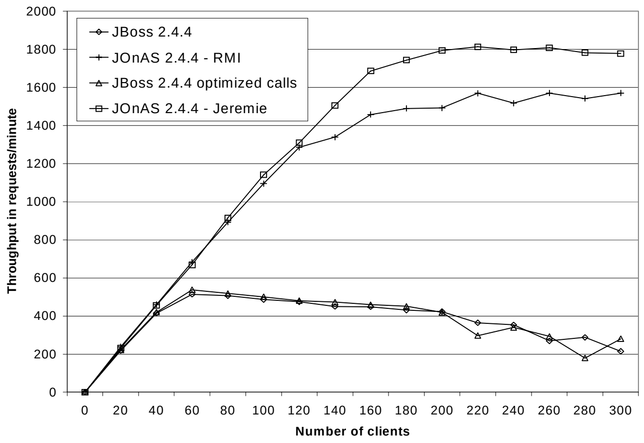

Figure 10. DAO separation with EB BMP implementation throughput in interactions per minute as a function of number of clients for the browsing mix using JBoss and JOnAS.

There is little difference between the CMP and BMP implementations. Both JBoss CMP configurations give comparable peak performance, with 534 and 559 interactions per minute reached with 80 clients for *JBoss* and *JBoss optimized calls*, respectively. JBoss BMP peak throughput is about 4% below the CMP version with 514 and 538 interactions per minute reached with 60 clients for *JBoss* and *JBoss optimized calls*, respectively. *JOnAS-RMI* peaks at 1670 interactions per minute with 300 clients using the CMP version, and at 1570 interactions per minute with 260 clients using the BMP version. The best results are achieved by *JOnAS-Jeremie* with 1858 interactions per minute with 200 clients using the CMP version and 1813 with 330 clients using the BMP version.

Figure 11 and figure 12 show the throughput in interactions per minute as a function of number of clients for the bidding mix for the CMP and BMP implementations, respectively.

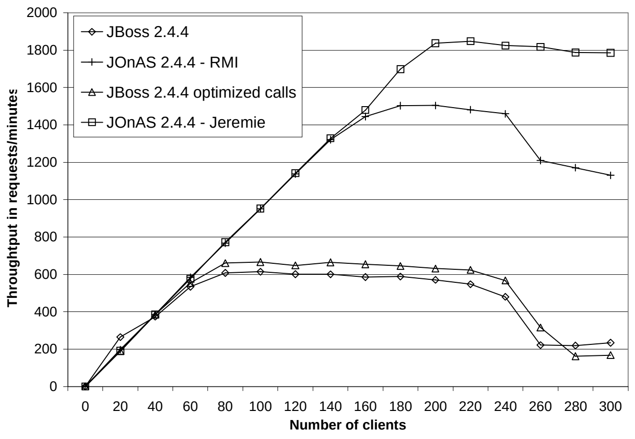

Figure 11. DAO separation EB CMP throughput in interactions per minute as a function of number of clients for the bidding mix using JBoss and JOnAS.

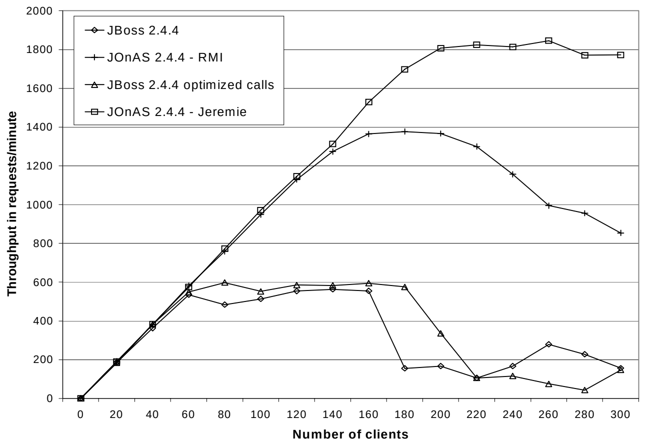

Figure 12. DAO separation EB BMP throughput in interactions per minute as a function of number of clients for the bidding mix using JBoss and JOnAS.

The ordering of the different configurations in terms of performance is the same as for the browsing mix. *JBoss* reaches a peak of 615 interactions per minute for 100 clients with CMP and 563 for 140 clients with BMP. There is an 8% improvement in

CMP when passing objects by reference, resulting in 666 interactions per minute for *JBoss optimized calls* with 100 clients. The BMP version improves by less than 6% (598 interactions per minute with 80 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 1504 interactions per minute for 200 clients with CMP and 1377 for 180 clients with BMP. *JOnAS-Jeremie* achieves 1848 interactions per minute with 220 clients for CMP. The results for BMP are almost the same, with a peak of 1846 interactions per minute with 260 clients.

We notice a drop in performance for *JOnAS-RMI* and both configurations of JBoss with 240 clients in the CMP version. The same thing happens for BMP but earlier and more abruptly for JBoss than for *JOnAS-RMI*. This sharp drop in performance occurs when the container is overloaded and transactions starts to timeout. After this point, it becomes very hard to obtain stable results.

The BMP and CMP implementations offer similar performance and have comparable behavior. Therefore, the drop in performance compared to the session beans implementation is not due to the container persistence service. On the contrary, the CMP version performs a little bit better than the hand-coded BMP version that cannot benefit from some internal container optimizations on lookups (see profiling analysis).

Rather than the container-managed persistence, it is the fine granularity of the interactions resulting from entity beans that is responsible for its performance being much lower than that of session beans. This granularity is a major issue for performance since each data access needs 2 network round-trips: one from the servlet to the entity bean and one from the entity bean to the database (unless there is a cache hit in the EJB server on this entity bean). Looking at the network statistics, we find that carrying the same amount of data requires about twice as many messages in EB, compared to SB.

Figure 13 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 session beans 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 the total execution time. Of that 40%, one fourth is due to reflection. In comparison, the *JOnAS* container uses much less CPU time (both in the container in general and in reflection in particular). JBoss's client stub optimization does not seem to work with entity beans. As a result, the communication time is slightly lower for *JOnAS-RMI* than for *JBoss*, *JOnAS-RMI* thus performs better both in terms of communication and container time, and has better overall throughput. *JBoss optimized calls* reduces the time spent in communication, resulting in some performance improvement over *JBoss*. The large amount of time spent in the container, however, results in inferior throughput compared to JOnAS, even with RMI. *JOnAS-Jeremie* shows a further improvement in throughput over *JOnAS-RMI*. Again, even though its container classes are more expensive than *JOnAS-RMI*, the gain in communications time leads to an overall improvement. The breakdown of the communication cost between TCP/IP and the layer above it (RMI or Jeremie) is the same as the one observed with session beans.

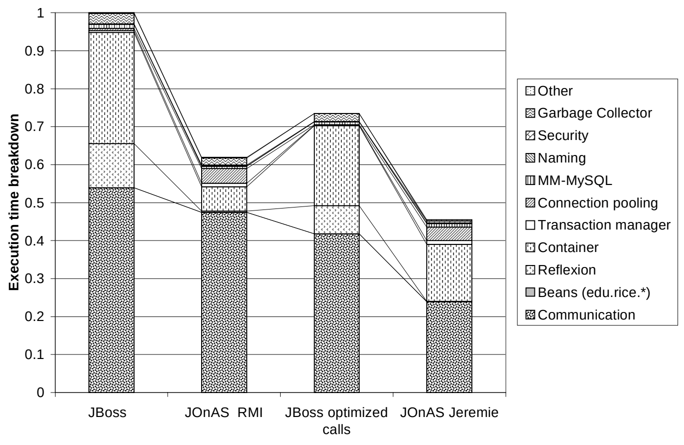

Figure 13. Execution time breakdown for the EB CMP implementation for the bidding mix at the peak point of the *JBoss* configuration.

Figure 14 shows the execution time breakdown for the EB BMP implementation for the bidding mix at 80 clients (the peak point of the *JBoss* configuration). The time spent in the bean code has almost doubled, but it is still very low with about 0.2% of the overall execution time. The counterpart is a slight decrease of the time spent in the container, but the results are very close to those obtained with EB CMP. Therefore, using container- or bean-managed persistence with entity beans has little influence on performance since container CPU usage remains almost the same. The contributions of various aspects of the implementation remains roughly the same as with entity beans and CMP.

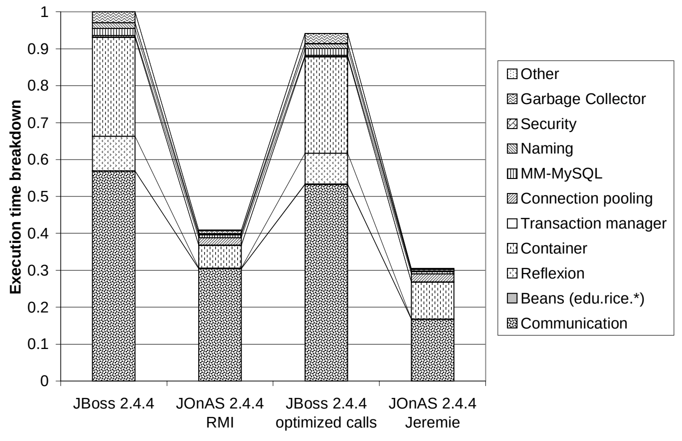

Figure 14. Execution time breakdown for the EB BMP implementation for the bidding mix at the peak point of the *JBoss* configuration.

The slight performance slowdown with EB BMP compared to EB CMP is due to the extra time spent in the naming service. When using BMP, lookups are less efficient with both containers than when using CMP. Naming with EB CMP represents 0.1% of overall execution time with JBoss and 0.2% with JOnAS. With

BMP, the naming service takes more than 1.5% of the total execution time.

In summary, unlike for the session beans version, the container design has the largest impact on performance for entity beans. Bean- or container-managed persistence offer similar performance demonstrating that performance slowdown compared to session beans is due to the excessively fine granularity data access exposed by entity beans. Optimized communications still improve performance but to a lesser extent.

6.4 Session façade implementation

Figure 15 presents the throughput in interactions per minute as a function of number of clients for the browsing mix using the 4 container configurations.

Due to the communication overhead between the session façade 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, compared to EB CMP. We do not report throughput for more than 200 clients, because *JBoss* becomes unable to handle the load and transactions abort on timeout.

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 16 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.

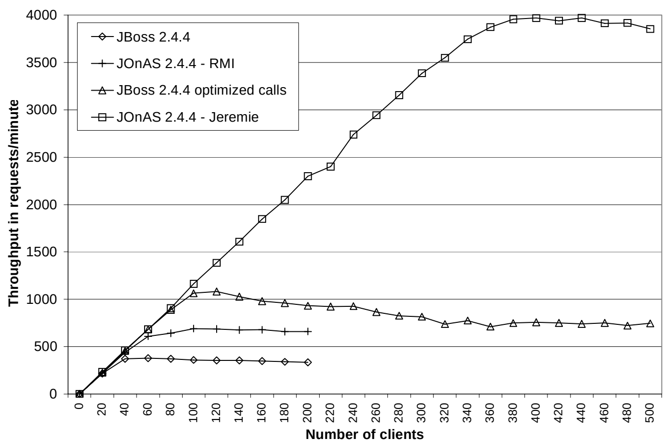

Figure 15. Session façade implementation throughput in interactions per minute as a function of number of clients for the browsing mix using JBoss and JOnAS.

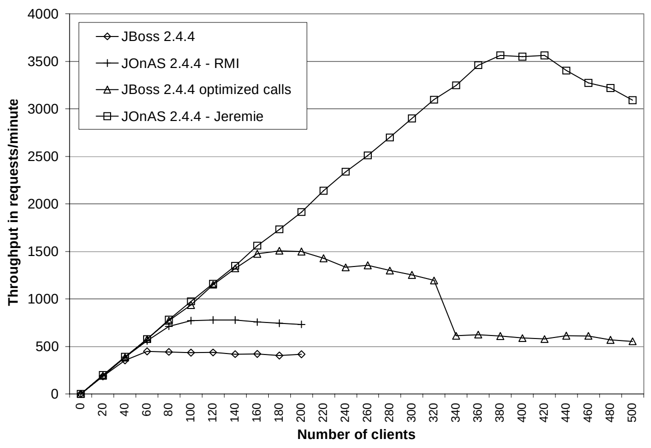

Figure 16. Session façade implementation throughput in interactions per minute as a function of number of clients for the bidding mix using JBoss and JOnAS.

JBoss optimized calls offers better performance with a peak at 1507 interactions per minute with 180 clients. However, response time dramatically increases under saturation. At 340 clients some transactions take more than 5 minutes to complete and are timed out by the transaction manager. Then, the number of completed interactions drops to 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 17 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.

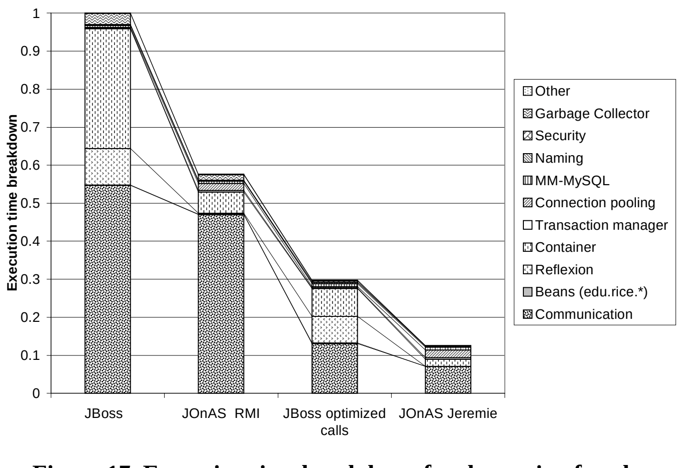

Figure 17. Execution time breakdown for the session façade implementation for the bidding mix at the peak point of the *JBoss* configuration.

There is little difference with the EB implementation for both *JBoss* and *JOnAS-RMI*. Communication time is the largest component of execution time, but the difference between the two configurations stems from differences in the container.

As the number of beans and interactions between beans increase, the time spent in reflection with the *JBoss optimized calls* configuration increases, as does the time spent in the container. The call optimization is visible in the reduction of the CPU utilization dedicated to communication. For the first time, we observe that more time is spent in the container than in communications.

JOnAS-Jeremie reduces the communication time even further. The container CPU time is also low, resulting in good overall throughput. The larger number of lookups on beans explains the time spent in the naming directory. Whereas RMI-based configurations have still a 71%/29% distribution of communication costs between TCP/IP and RMI, with Jeremie the distribution becomes 65.8% for Jeremie versus 34.2% to TCP/IP. This is due to the fact that Jeremie's optimization for local calls is heavily exercised in this implementation, resulting in more computation in the Jeremie layers and less communication going through TCP/IP.

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.

6.5 EJB 2.0 local interfaces

Figure 18 shows the throughput in interactions per minute as a function of number of clients for the browsing mix using 3 of the 4 container configurations. We do not present the results for JBoss without the optimized communication layer, because we cannot disable this optimization. Note that this experiment uses different containers, namely JBoss 3.0 and JOnAS 2.5.

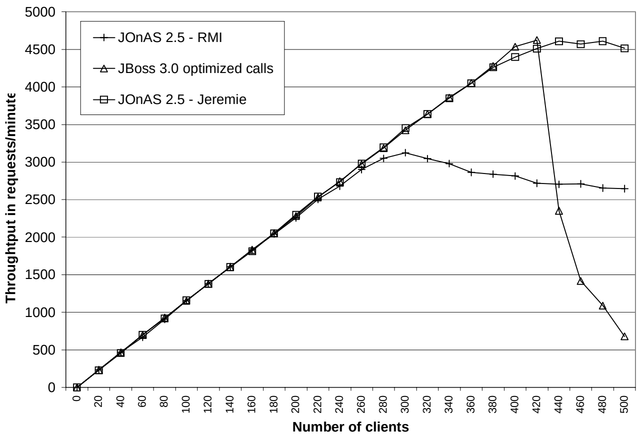

Figure 18. EJB 2.0 local interfaces implementation throughput in interactions per minute as a function of number of clients for the browsing mix using JBoss and JOnAS.

All three configurations give better peak performance when using EJB 2.0 local interfaces than with the EJB 1.1 session façade version. *JOnAS–RMI* improves by a factor of 4.5 and peaks at 3125 interactions per minute with 300 clients. *JBoss optimized calls* and *JOnAS–Jeremie* give almost the same performance until 420 clients. *JBoss optimized calls* reaches its peak point at 4623 interactions per minute with 420 clients (4.3 times better than the EJB 1.1 implementation), but afterwards performance drops to 678 with 500 clients. This big drop in performance is due to the way the container handles transaction timeouts. As soon as the container has to rollback transactions, performance drops sharply. *JOnAS–Jeremie* peaks at 4605 interactions per minute for 480 clients, and remains stable for larger numbers of clients. As Jeremie already optimizes the local calls, we only notice a 16% improvement compared to the EJB 1.1 implementation.

Figure 19 reports the throughput in interactions per minute as a function of number of clients for the bidding mix. Results are similar to those obtained with the browsing mix. *JOnAS-RMI* achieves 2837 interactions per minute for 320 clients. *JBoss optimized calls* peaks 3641 interactions per minute with 380 clients, but performance falls for higher loads, to around 500 interactions per minute. *JOnAS-Jeremie* still offers more stable behavior and scales further achieving 4228 interactions per minute with 460 clients.

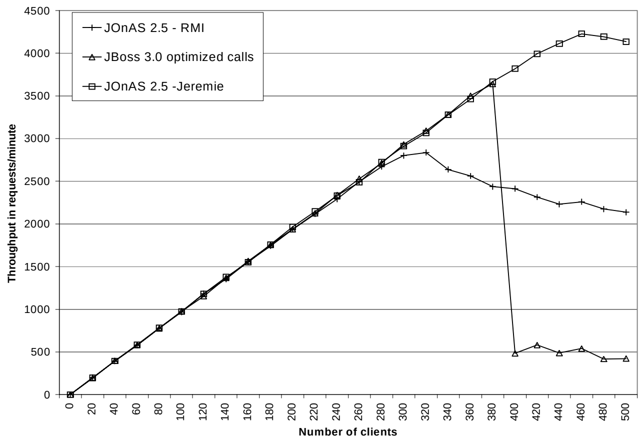

Figure 19. EJB 2.0 local interfaces implementation throughput in interactions per minute as a function of number of clients for the bidding mix using JBoss and JOnAS.

The bidding mix involves more transactions, and therefore the JBoss container collapses earlier than with the browsing mix. However, the J2EE servers using communication layers relying on RMI (*JOnAS-RMI* and *JBoss optimized calls*) get a significant factor of improvement using EJB 2.0 local interfaces. Optimized communication layers such as Jeremie can still benefit from EJB local 2.0 interfaces but the improvement remains below 19%.

Figure 20 shows the execution time breakdown for the EJB 2.0 local interfaces implementation for the bidding mix with 320 clients (the peak point of the *JOnAS-RMI* configuration). A major change appears in the new JBoss container where reflection now only represents a small fraction of the total execution time.

Communication remains the main CPU consumer for RMI-based configurations. Jeremie still shows the gain of an optimized communication layer. The communication time distribution between TCP/IP and RMI is now 79.5% and 20.5% for both *JOnAS-RMI* and *JBoss optimized calls*. *JOnAS-Jeremie* is more balanced with 48% for TCP/IP and 52% for Jeremie.

Due to the large number of beans and higher throughput, now a large fraction of time is spent in connection pooling that is less efficient with JOnAS than with JBoss. This increase of connection pooling time results in a decrease of time spent in the database driver when compared to the session façade implementation using EJB 1.1.

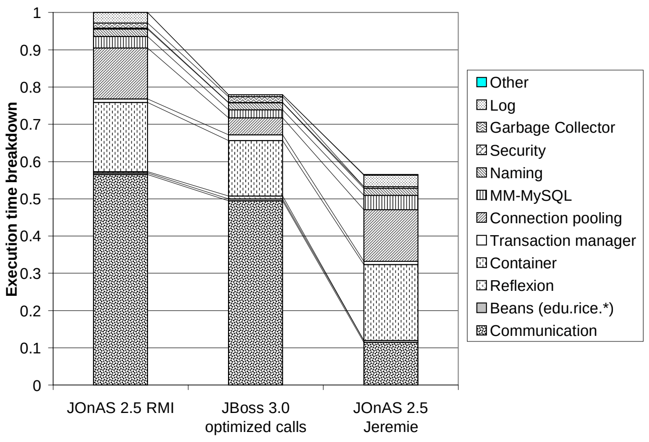

Figure 20. Execution time breakdown for the EJB 2.0 local interfaces implementation for the bidding mix at the peak point of the *JOnAS-RMI* configuration.

The new logging mechanism used in JOnAS 2.5 (a wrapper on top of log4j [17]) results in noticeable CPU consumption.. JBoss uses log4j directly and only spends 0.6% of its execution time in logging.

Figure 21 shows the execution time breakdown for the JBoss 3.0 container after its peak point. The transaction manager clearly becomes the largest component of the overall execution time, followed by communications and the container

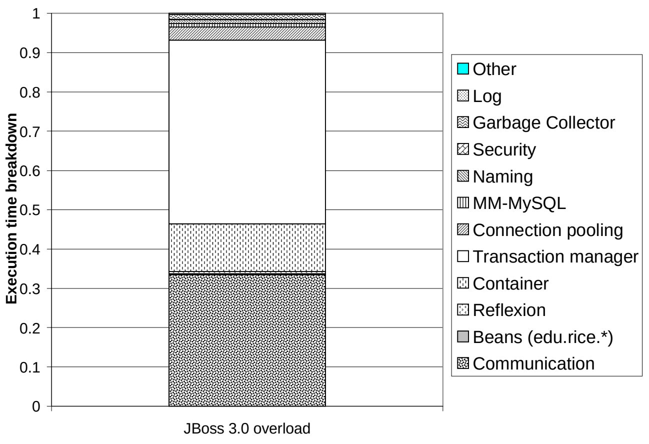

Figure 21. Execution time breakdown of the *JBoss 3.0* configuration for the EJB 2.0 local interfaces implementation for the bidding mix after the peak point.

In summary, when using a session façade pattern, EJB 2.0 local interfaces result in reduced communication overhead and greatly improve RMI-based configurations. However, optimized communication layers such as Jeremie still offer more scalable performance and need less CPU resources. The results show that every component of an EJB container is important to provide reliability and high performances. The JBoss transaction manager is the bottleneck after the peak point resulting in a dramatic collapse of performance.

6.6 JDK 1.4

Sun has introduced many improvements in the J2SE (Java 2 Standard Edition) version 1.4. The Performance and Scalability Guide [23] claims that reflective method invocation has been improved by a factor of 20. Other enhancements include JNI method invocation, object serialization and thread management. Unfortunately, we observe a lot of instability in our measurements with JDK 1.4 with Sun's JVM version 1.4.0_01 for Linux. We are not able to obtain reproducible results with JOnAS and JBoss 3.0. We are, however, able to perform experiments with JBoss 2.4.4, for which reflection is a significant portion of the execution time.

6.6.1 Entity beans

Figure 22 reports the throughput in interactions per minute as a function of number of clients for the browsing mix on the EB CMP implementation (see section 6.3) using JDK 1.3 and 1.4 with the *JBoss 2.4.4. optimized calls* configuration. The peak point with JDK 1.4 improves by less than 2.6%, achieving 587 interactions per minute with 80 clients, whereas JDK 1.3 achieves 572 interactions per minute with 60 clients. Except for the peak point, overall performance is worse with JDK 1.4, and we notice a significant drop of the throughput starting with 200 clients.

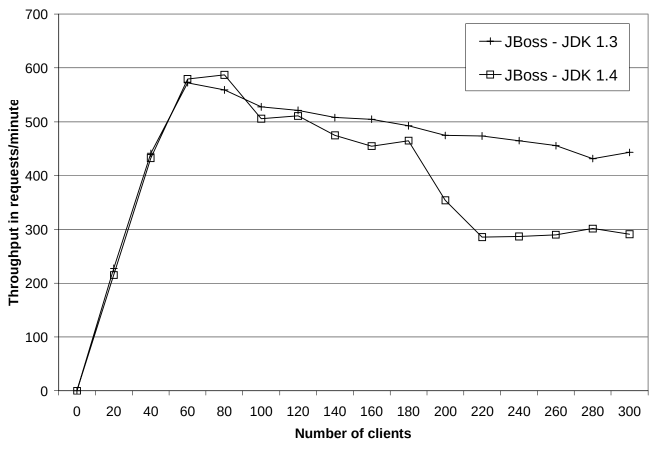

Figure 22. EB CMP implementation throughput in interactions per minute as a function of number of clients for the browsing mix using JBoss 2.4.4 for JDK 1.3 and 1.4.

Figure 23 reports the throughput in interactions per minute as a function of number of clients for the bidding mix for the EB CMP implementation using JDK 1.3 and 1.4 with the *JBoss 2.4.4. optimized calls* configuration. The peak throughput with JDK 1.4 improves by less than 12%, to 746 interactions per minute with 100 clients. The JDK 1.3 configuration peaks at 666 interactions per minute with the same number of clients. Just after the peak point, JDK 1.4 performance drops below the throughput obtained with 1.3.

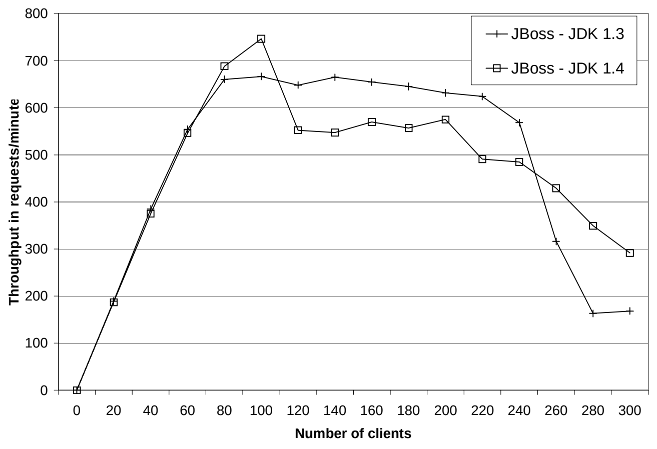

Figure 23. EB CMP implementation throughput in interactions per minute as a function of number of clients for the bidding mix using JBoss 2.4.4 for JDK 1.3 and 1.4.

Figure 24 shows the execution time breakdown with 100 clients (the peak point with JDK 1.3) for the bidding mix using the *JBoss optimized calls* configuration with JDK 1.3 and 1.4. Reflection time has been reduced with JDK 1.4, but container time has increased. This increase is mainly due to the timer tasks being less efficient in JDK 1.4. As a result, overall container time, including reflection, has only decreased marginally, resulting in only a small improvement in throughput.

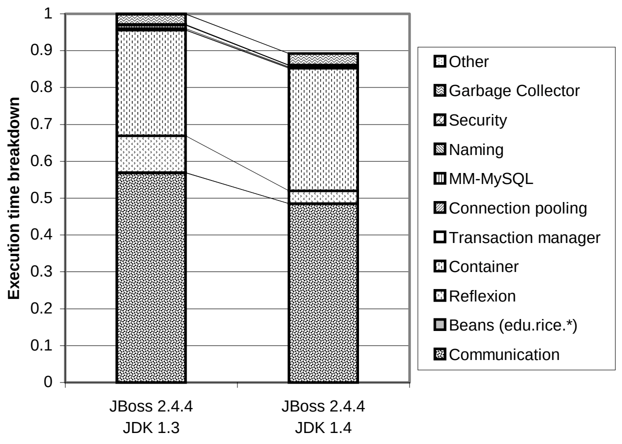

Figure 24. Execution time breakdown for the EB CMP implementation for the bidding mix at the peak point of the *JBoss optimized calls* configuration using JDK 1.3.

In summary, JDK 1.4 does not improve the overall performance of the EB CMP implementation with JBoss. We expect similar results with the EB BMP implementation since its profiling results are very similar to those for EB CMP. JDK 1.4 reduces the reflection cost compared to JDK 1.3, but the container cost rises mainly due to inefficiencies in timer task management.

6.6.2 *Session façade beans*

Figure 25 reports the throughput in interactions per minute as a function of number of clients for the browsing mix for the session façade implementation (see section 6.4) using JDK 1.3 and 1.4 with the *JBoss optimized calls* configuration.

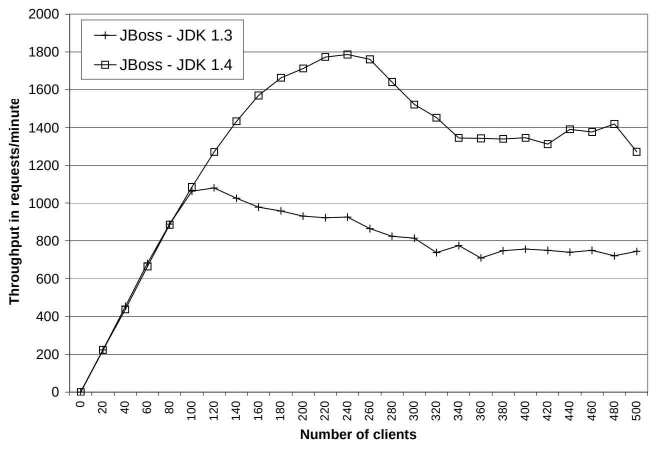

Figure 25. Session façade implementation throughput in interactions per minute as a function of number of clients for the browsing mix using JBoss 2.4.4 on JDK 1.3 and 1.4.

JDK 1.4 peak throughput is 1787 interactions per minute with 240 clients, 65% better than the 1081 interactions per minute with 120 clients obtained with JDK 1.3.

Figure 26 reports the throughput in interactions per minute as a function of number of clients for the bidding mix on the session façade implementation (see section 6.4) using JDK 1.3 and 1.4 with the *JBoss optimized calls* configuration. JDK 1.4 peak performance is 10% better than JDK 1.3 at 1657 interactions per minute with 200 clients, whereas JDK 1.3 achieves 1507 interactions per minute with 180 clients. JDK 1.3's performance drops to just above 600 interactions per minute with 340 clients, but JDK 1.4's performance remains more stable between 1172 and 949 interactions per minute.

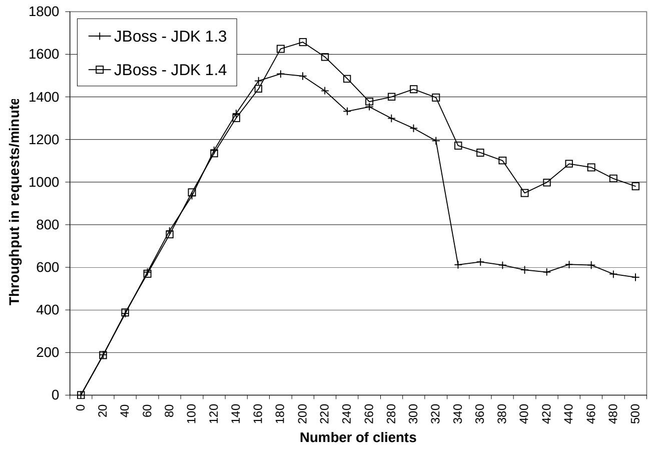

Figure 26. Session façade implementation throughput in interactions per minute as a function of number of clients for the bidding mix using JBoss 2.4.4 using JDK 1.3 and 1.4.

Figure 27 shows the execution time breakdown with 180 clients (the peak point with JDK 1.3) for the bidding mix using the *JBoss optimized calls* configuration with JDK 1.3 and 1.4. We observe less gain in reflection compared to EB CMP. Once again. container time has increased. Therefore, overall container time including reflection has only decreased by a moderate amount, resulting in a modest throughput improvement. We have not been able to explain the larger improvement for the bidding mix.

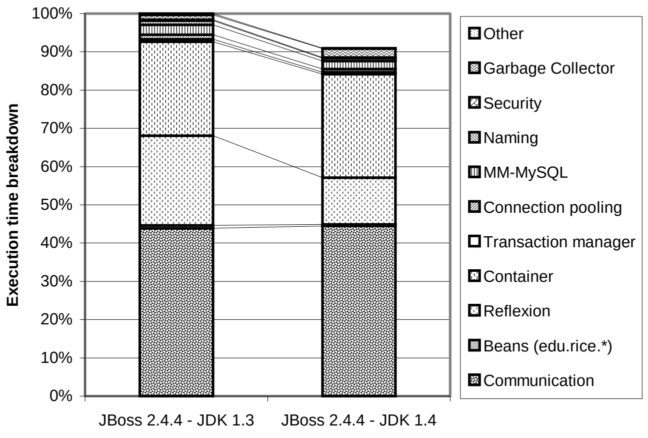

Figure 27. Execution time breakdown for the session façade implementation for the bidding mix at the peak point of the *JBoss optimized calls* configuration using JDK 1.3.

6.7 Summary

Figure 28 and figure 29 summarize the peak throughput obtained for the different application implementation methods and container configurations for the browsing and the bidding mix, respectively.

The session beans 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.

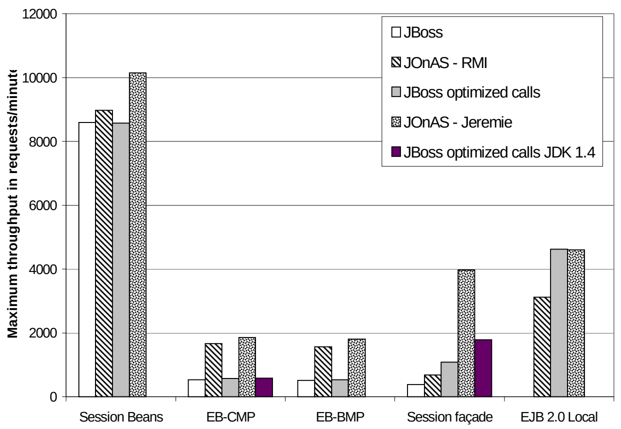

Figure 28. EJB implementations maximum throughput in interactions per minute for the browsing mix.

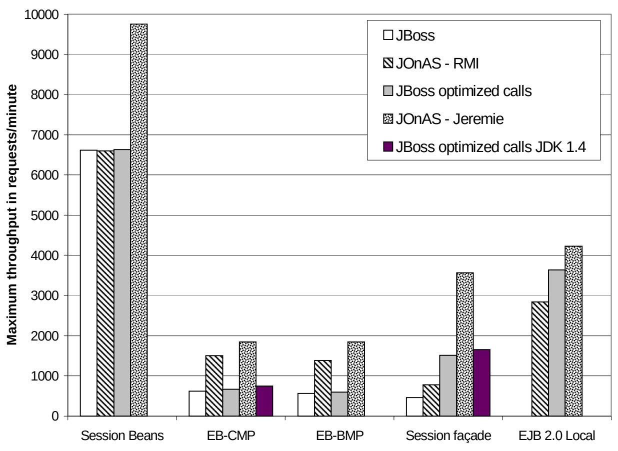

Figure 29. EJB implementations maximum throughput in interactions per minute for the bidding mix.

The DAO separation with entity beans implementation gives the least scalable results. Container-managed persistence gives slightly better results than bean-managed persistence. The poor performance is therefore not due to the container implementation of persistence. Instead, the excessively fine-grain access exposed by the entity beans to the servlets causes too many remote interactions. This implementation, however, shows that container design has a significant impact on the performance of entity beans. 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 communication optimizations are not sufficient to mask the overhead of reflection in the container. JDK 1.4 reduces the cost of reflection, but increases the cost of the container. Therefore, performance does not improve much. Only the combination of pre-compiled container classes and an optimized communication layer such as Jeremie allows for scalable performance with session façade beans.

The use of EJB 2.0 local interfaces reduces the cost of communications, because intra-JVM calls do not go through the communication layers. This implementation offers more scalable results. Both JBoss and JOnAS EJB 2.0 compliant containers make little use of reflection and offer a significant improvement for RMI-based configurations. Optimized communication layers such as Jeremie can also benefit from local interfaces but to a lesser extent.

The bean code written by the programmer represents at most two percent of the overall execution time. Most of the bean code consists of calls to middleware services. 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, for instance, 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 [21] 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 TPC-W [25] 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 application scalability, analyzing the container and communication layer designs. Others have given guidelines for EJB server comparison [11], but they use the EJB 1.0 specification and they do not propose an application to perform the comparison. We have made available the application, container configurations and experimental 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) [27]. 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 communication layer optimizations. 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, for example with the session façade pattern, that reflection can become a real issue for scalability.

The EJB CMP 2.0 specification [22] addresses the issue of fine-grain access exposed by the entity beans and provides a specific EJB QL query language for complex finder queries. This evaluation will be part of our future work when the implementation becomes available. We also plan to experiment with a Message Driven Beans implementation of RUBiS to evaluate the performance and scalability of J2EE applications using asynchronous communications.

Another solution to achieve scalability is to use a cluster. Major J2EE vendors implement such as BEA [7] or IBM [10] use clustering to achieve scalability and high availability. We plan to evaluate clustering when it becomes available in the open-source containers we use for our evaluation.

8. CONCLUSIONS

We have experimented with several EJB 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 full experimental reports including performance charts and resource usage, are 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 a servlets-only implementation. 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 session beans, because communication costs dominate, but 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. Although reflection is cheaper in JDK 1.4, it does not improve the performance of entity beans, because of other inefficiencies.

Container design and the cost of local communication 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 offers scalable performance. Reflection cost increases with the number of beans, quickly resulting in a bottleneck. JDK 1.4 reduces the cost of reflection but increases the time spent in the container classes. The end result is performance that remains inferior to that obtained with pre-compiled container classes and fast local communication. EJB 2.0 local interfaces avoid the communication layers for local communications and allow RMI-based configurations to scale better.

9. ACKNOWLEDGMENTS

We are grateful for the people of the JBoss and JOnAS communities for the useful help and information they provided us. We also thank the anonymous reviewers for their valuable comments that helped us to improve the quality of this article.

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 Microsystems 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] Byte Code Engineering Libray (BCEL) – http://jakarta.apache.org/bcel/.

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

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

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

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

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

[12] 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.

[13] eBay – http://www.ebay.com/.

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

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

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

[17] Log4j – http://jakarta.apache.org/log4j/docs/index.html.

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

[19] OptimizeIt Profiler – http://www.borland.com/optimizeit/.

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

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

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

[23] Sun Microsystems – Java 2 Platform Standard Edition 1.4, Performance and Scalability Guide – http://java.sun.com/j2se/1.4/performance.guide.html, 2002.

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

[25] Owen Taylor – J2EE Data Access Objects – The Middleware Company - http://www.middleware-company.com/documents/DAOPattern.pdf, 2002.

[26] Transaction Processing – http://www.tpc.org/.

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