J2EE Server Scalability through EJB Replication

Sylvain Sicard
INRIA Rhônes-Alpes
Zirst – 655 avenue de l'Europe
38334 St Ismier CEDEX - France
Sylvain.Sicard@inrialpes.fr

Noel De Palma
INRIA Rhônes-Alpes
Zirst – 655 avenue de l'Europe
38334 St Ismier CEDEX - France
Noel.DePalma@inrialpes.fr

Daniel Hagimont
INRIA Rhônes-Alpes
Zirst – 655 avenue de l'Europe
38334 St Ismier CEDEX - France
Daniel.Hagimont@inrialpes.fr

ABSTRACT
With the development of Internet-based business, Web applications are becoming increasingly complex. The J2EE specification aims at enabling the design of such web application servers. These servers have to ensure scalability and availability of the supported applications. Scalibility can be achieved using replication techniques or partitionning techniques. The aim of this paper is to compare these approaches. In a J2EE web application server, one important component is the EJB tier. In this context, the JOnAS web application server provides an example of EJB replication system called CMI (Cluster Method Invocation). In a first step, this paper presents a performance evaluation of CMI. It then introduces incrementally an alternative scheme based on partitionning and shows the performance benefits compared to CMI.

Categories and Subject Descriptors
D.4.8 [**Operating Systems**]: Performance – *Measurements, Monitors.*

General Terms
Algorithms, Measurement, Performance, Design, Reliability, Experimentation.

Keywords
J2EE, replication, partition, scalability, EJB.

1. INTRODUCTION
The J2EE specification [14] aims at enabling the design of complex web application servers, structured in several tiers. A J2EE application server is generally composed of four tiers which can execute on different machines, as illustrated in Figure 1:

- The web tier executes client requests (e.g. Apache [16]): a reference to a static web page is resolved by the web server , a reference to a dynamic page is forwarded to the Servlet tier.
- The Servlet tier (e.g. Tomcat [5]) generates a web page on the fly (the creation of the web page is dynamic), from data which can be requested to the EJB tier.
- The EJB tier (e.g. JOnAS [6]) includes the functional code of the application which computes the data requested by the Servlet tier. The EJB server interacts with a database server which manages the persistent data of the application.
- The database tier (e.g. MySQL [11]) which manages persistent data.

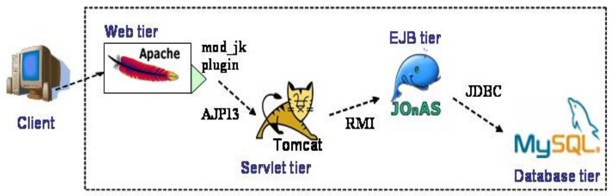

Figure 1 : Architecture of a typical J2EE application server

The overall J2EE application server must be scalable. The scalability is defined as its ability to deal with an arbitrary number of client requests in a reasonable time (for each individual client).

Scalability can be achieved by replication of the different tiers on a cluster of machines (also called *clusterization*). Currently, the replication scheme which is always employed for J2EE servers consists in replicating all tiers: each tier (e.g. JOnAS) can be replicated on several machines and it is entirely copied on these machines. Client requests are routed toward one of the replicas in order to balance the load, a random selection of the replica being generally used (Round-Robin strategy). This solution raises an important issue related to data consistency. A replicated tier (in particular an EJB server) can include a modifiable state which must be kept consistent. This has a strong incidence on the clusterization of a web application server.

Scalability can also be achieved using patitionning technique. The principle of partitioning is to deploy on each server only a fragment of the application, i.e. a subset of its objects (without any intersection between the subsets). Therefore, an object is located in one unique partition and this partition is deployed on one unique server. Consequently, data are not replicated and consistency is not an issue anymore.

This article studies the scalability of a J2EE application server by replication of the EJB tier and and compare this approach to the partitioning technique. In a first step, we present the context and the motivations for our work (Section 2). After a description of our experimentation environment (Section 3), we report on the performance of different J2EE server configurations, which reveals the advantages and drawbacks of these configuration choices (Sections 4 and 5). We then present the design of an EJB hybrid replication scheme and a performance evaluation which demonstrates its benefits.

2. Context and motivations

The main goal of the clusterization of J2EE servers is to bring scalability and availability for web applications. Regarding scalability, a web application must be able to serve millions of requests per day. This is generally enabled by massively replicating all the J2EE tiers. An example of such a clusterized architecture is presented in Figure 2.

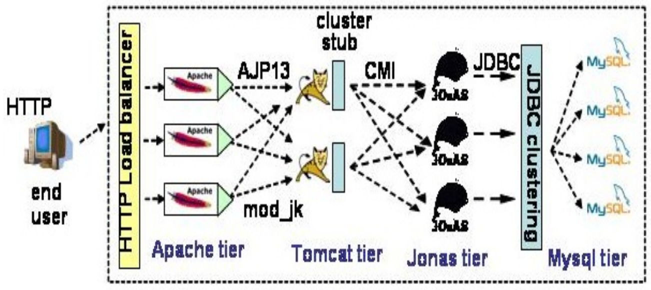

Figure 2 : Clusterized J2EE Architecture

In this example, the web tier is composed of a set of replicated Apache web servers. The Apache servers only include a read-only state, so they don't bring any data consistency problem. Replication can be made transparent to the client thanks to the hardware or software techniques, such as: Level-4 switch (where a dedicated router can simultaneously distribute up to 700000 TCP connections towards different servers), TCP handoffs (where a frontal server establishes TCP connections and delegates treatments to slave servers), or Round-Robin DNS (where a DNS server periodically change the IP addresses associated with the name of the web site).

The Servlet tier is composed of replicated Tomcat Servlet servers. The Apache tier distributes incoming requests between Tomcat servers thanks to the AJP13 connector and the Apache mod_jk plugin which implements the load balancing strategy between the Tomcat replicas. The Tomcat replicas can maintain a modifiable global state. This state is kept consistent thanks to a group communication protocol.

The EJB tier is composed of replicated JOnAS EJB servers. The EJB servers manage a set of beans (Java objects) which include a modifiable state. In this example, replication and consistency are enforced by CMI (Cluster Method Invocation, the EJB clusterization tools provided by JOnAS). Each Tomcat server interacts with EJB servers through cluster-stubs (similar to RMI stubs, but in CMI) which distribute the load between the replicas following a Round-Robin strategy.

An EJB server, in its treatments, may have to interact with the database tier which manage persistent version of beans. These interactions being very costly, the EJB server manages a cache of beans which keeps a local copy of beans that were fetched from the database. However, if the EJB server is replicated, the same bean may be copied in the caches of two different EJB server replicas (see Figure 3), thus leading to a consistency problem. CMI prevents this problem by disabling cache management in the EJB servers. Any (committed) bean modification by one EJB server is applied on the database and subsequently read from the database by another EJB server which shares the same bean. Shared beans are therefore synchronized by the database.

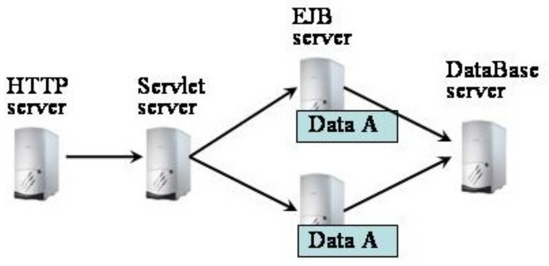

Figure 3 – Data consistency problem

The database tier is composed of a set of replicated Mysql database servers. The databases are kept consistent thanks to C-JDBC [2]. The EJB servers are connected to the databases via a C-JDBC controller which balances the load between the database replicas, but also maintains consistency between the replicas by consistently propagating updates on the replicas.

3. Environnement

For our experimentations, we used the Apache HTTP server (version 1.3.31), the Tomcat Servlet server (version 3.3.1a), the JOnAS EJB server (version 1.4.2), the MySQL database server (version 4.0.20). In order to ensure that the database tier does not saturate before the EJB tier, we used the C-JDBC middleware (version 1.0) to over-allocate machines at the database tier.

To evaluate the performance of our J2EE application server, we used the RUBiS benchmark [1]. RUBiS is a prototype of web application which models an auction system similar to eBay. RUBiS implements all the basic functions of this type of web site. Among the most important ones are browsing items, bidding, buying or selling items, leaving comments on other users rating or consulting one's user page. The system is sized according to observations found on the eBay web site.

RUBiS is a full J2EE application which, once deployed on a hardware configuration, allows measurement of the web server under a given workload. RUBiS provides different implementations of the same application in order to stress the different tiers of the J2EE architecture. We used an EJB based implementation, the Session Facade Bean pattern and a CMP persistence management, as it sets more load on the JOnAS server and less on the Tomcat server. RUBiS includes a load injector which emulates clients which connect to the web site from a web browser. The behavior of the clients is modeled with Markov chains. The clients emulator also performs monitoring of all the involved machines. In the reported experiments, we are mainly interested in the overall web site throughput in terms of requests per second, when stressful load is generated. We generate this load by creating a large number of clients (eventually of different machines).

4. EJB Server without replication

In this section, we are interested in studying the performance of a single EJB JOnAS server. As we have seen in Section 2, replication may be conflicting with caching. We are therefore interested in studying the effect of caching on the performance of an EJB server.

We recall that in the EJB specification, application components which are persistent are called *entity beans*; only entity bean can be kept in the cache. *Beans* can be created thanks to a *factory*. A factory provides both the means to create a bean and to obtain a reference to an existing one from a symbolic name. Each factory in an EJB server maintains its own cache. Therefore, there's one cache per bean factory and a one cache only includes beans created by its associated factory.

In a J2EE application, a deployment descriptor specifies for each factory whether its beans can be shared or not (between different replicas). If a bean can be shared, it will not be maintained in its factory's cache and it will be systematically read/written from/to the database for each access to the bean (actually for each transaction). If the bean cannot be shared, it will only be loaded from the database at first invocation and kept in the cache for further accesses.

4.1 Performance with cache

In this section, we report the performance obtained with the RUBiS benchmark when the cache is enabled. In this experiment, the application descriptor specifies that beans cannot be shared. The cluster configuration we used for this measurement includes 7 machines as described in Figure 4. In this experiment, the load on the database tier is not very important and does not require a cluster of databases (using C-JDBC), but we settled such a database cluster in order to be consistent with other experiments (described further in the paper). Therefore, throughputs and latencies observed in all our experiments can be compared.

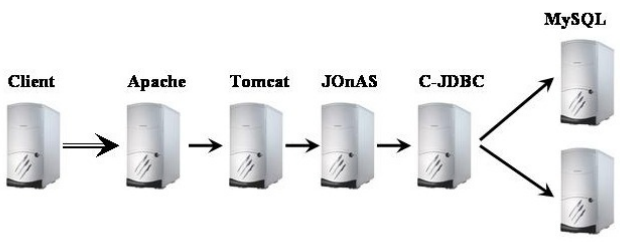

Figure 4 – Configuration for caching measurements

Figure 5 presents the results of this performance evaluation. It shows the throughput of the overall J2EE server (in number of requests per second) according to the number of clients. We observe that as long as none of the servers in the configuration saturates, the throughput increases linearly according to the injected load. The maximal throughput is reached for about 160 clients. This maximum corresponds to the saturation of the CPU resource on the machine which hosts the JOnAS server.

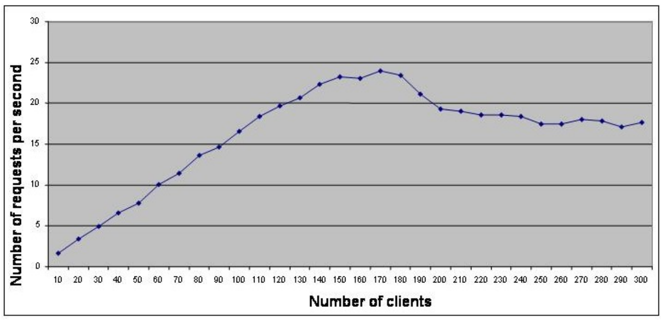

Figure 5 – Throughput with one EJB server and cache enabled

4.2 Performance without cache

The goal is here to reproduce the same experience, but with cache disabled on the EJB server, in order to quantify the impact of caching on performance. In this configuration, each access to a bean involves an access to the database.

Figure 6 presents the results of this evaluation. We observe that overall, the behavior of the server is similar as in the previous experience, except that here, the CPU resource saturates for a number of clients close to 80. We don't report results for more than 210 clients because at this load level, the EJB server was so saturated (including errors of many kinds) that results were unstable.

These two experiments show the impact of EJB caching on the performance of the overall J2EE server. Thanks to EJB caching, beans are kept locally on the EJB server, which reduces interactions with the database tier. In order to be efficient, a cache system must have a good hit rate. Thanks to complementary experiments, we measured that our hit rate in the JOnAS cache was about 86%, which is rather good, but is however application dependent;

The results obtained here show that caching at the level of the EJB server can have a very significant impact on the overall performance of the J2EE server. In a context where we aim at replicating the EJB server, and as replication may be conflicting with caching, it is crucial to find the best trade-off.

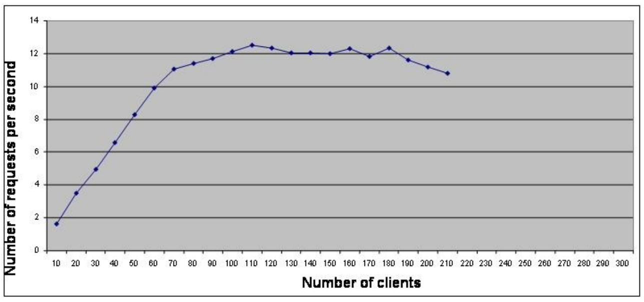

Figure 6 - Throughput with one EJB server and cache disabled

5. Replication of the EJB server with CMI

5.1 CMI without cache

The goal of this experience is to measure the performance of the EJB replication system based on CMI (which disables cache), and to compare it with those of a single EJB server with cache enabled (which was given in Section 4.1). We aim at showing that the performance benefits from replication can be counterbalanced by the disabling of the cache. This experience will also provide us a reference performance level, with which we can compare in further experiments.

CMI is a sub-component of JOnAS which implements an evolution of RMI-JRMP (Java Remote Message Protocol). CMI balances the invocation load between a set of JOnAS servers, thus providing scalability.

The invocation of a method on a bean from a servlet is performed as follows. The client queries a JNDI [15] name server to obtain from a symbolic name a reference (stub) to a factory. This factory is itself a name server for the beans it manages. The client can then query this factory to obtain a reference (stub) to a bean. The client can then invoke a method on that bean.

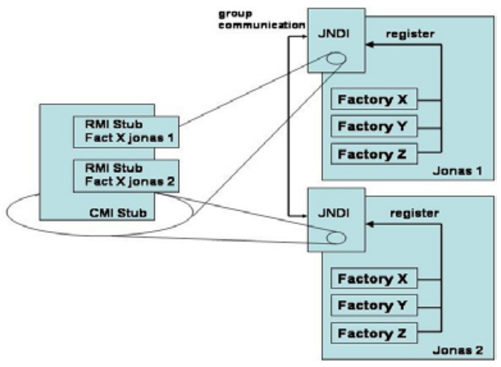

Figure 7 – Principle of CMI

The principle of CMI is illustrated in Figure 7. Each JOnAS server deploys all the bean factories of the application (all the factories are replicated on each EJB server). The JNDI server is replicated on each JOnAS server and the replicas are synchronized using group communication in order to register one CMI factory stub for each factory. A CMI factory stub includes all the RMI stubs of the factory replicas. When a client queries the JNDI server, it can be answered by any of the JNDI replicas and obtains a CMI factory stub. When a factory is queried using a CMI stub, the CMI stub selects one of the RMI stubs and uses it with a normal RMI invocation. The load balancing is implemented by the stub election algorithm (round robin). A factory replica on one EJB server returns a RMI stub which references a replica of the bean in that EJB server (this bean is eventually loaded from the database tier).

With the CMI scheme, all the factories (including the cache in each factory) are replicated. Thus, a bean can be loaded in two different EJB servers and the EJB server does not implement a distributed cache algorithm to maintain consistency between different bean copies. Therefore, using CMI requires disabling caching. This implies an increase of the number of interaction with the database tier. For each access to a bean, the state of the bean must be loaded from the database, and written to the database if modified. Functionally, CMI implements a sort of RAID-1 architecture where the EJB server is replicated and requests are distributed over the replicas.

For the evaluation of CMI, the configuration we used (Figure 8) is the same as the one used to evaluate caching (Figure 4), except that we replicate the JOnAS server.

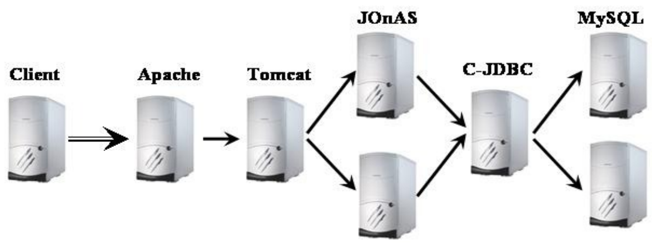

Figure 8 – Configuration for CMI measurements

5.2 Performance

Figure 9 reports the results. We can see that in this experience, the system reaches a maximal throughput for a number of clients close to 160 with a throughput of 23 requests per second. This experience shows that with the allocation of two nodes for the EJB tier, CMI can only provide the same throughput as one single EJB server which is cache enabled.

These results are fully consistent with those previously reported for the performance of caching. The maximal throughput in this evaluation (22 req/s) is twice the one obtained with one EJB server cache disabled (Section 4.2 – 12 req/s). This evaluation shows that the scalability of CMI is rather good.

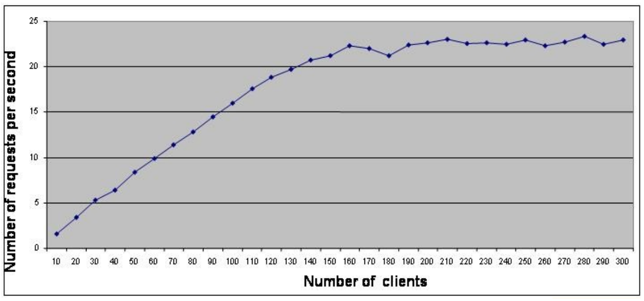

Figure 9 - Throughput with two EJB servers, CMI and cache disabled

6. Factory partitioning

6.1 Description

In this section, we consider a solution based on partitioning and we compare it to CMI. With CMI, each JOnAS server deploys the whole application. The principle of partitioning is to deploy on each JOnAS server only a fragment of the application, i.e. a subset of its beans (without any intersection between the subsets). Therefore, a bean is located in one unique partition and this partition is deployed on one unique EJB server. Consequently, beans cannot be replicated and consistency is not an issue anymore. Bean caching can therefore be activated in all EJB servers. In the following experiment, we used factory partitioning, i.e. each factory is deployed on a unique JOnAS server.

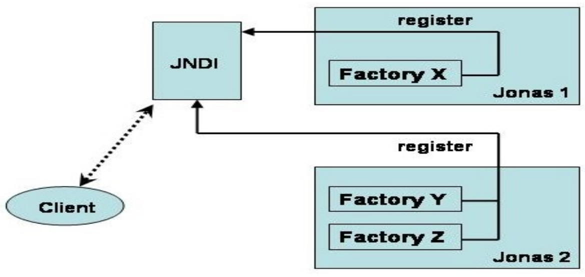

Figure 10 – Principle of factory partitioning

To implement this solution, we need to provide servlets the ability to locate the EJB servers where factories are deployed. This location is implicitly managed in the stubs that factories register in the JNDI naming service, as a factory stub (which is a RMI stub) is able to route an invocation towards the EJB server which hosts the factory. By configuring an architecture where all the EJB servers share the same JNDI service (Figure 10), factory location is implicit. In the evaluation configuration, we ran a JNDI server on a separate machine. Functionally, this solution implements a sort of RAID-0 architecture where the EJB server is replicated but the replicas are not equipotent. In this approach, availability is ensured by redeploying on a new machine beans located on a faulting machine. The transaction service enforces consistency.

6.2 Performance

Our evaluation has been conducted with the same configuration as the one used to evaluate CMI (Figure 8).

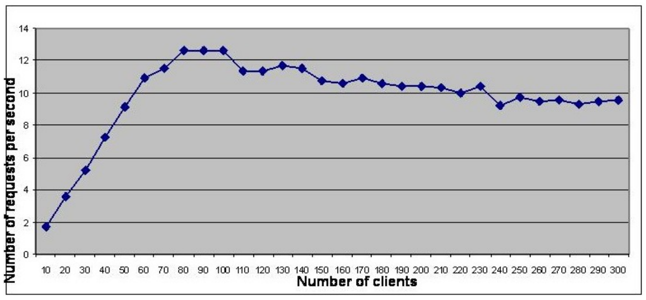

Figure 11 - Throughput with two EJB servers, partitioning and cache enabled

As we can see in Figure 11, the maximal throughput is obtained for 90 clients with a throughput of 13 requests per second. These results are quite disappointing. The maximal throughput is almost half the one obtained with CMI with the same hardware configuration.

To analyze more precisely these results, let's examine the differences between CMI and the proposed solution. The cost to locate a bean from a servlet through the JNDI service should be the same in both cases (a remote invocation in both cases).

With CMI, each JOnAS server integrates its own JNDI service, but in our solution the JNDI service is implemented as a server on a separate machine. Then, each request to JNDI from a bean within JOnAS requires a remote invocation in our solution, while it is a local invocation in the case of CMI.

AS shown in Figures 12 and 13, the number of remote communications is more important in our partitioning solution. With CMI, as soon as a session started in an EJB server, all invocations (to JNDI, a factory or a bean) are local.

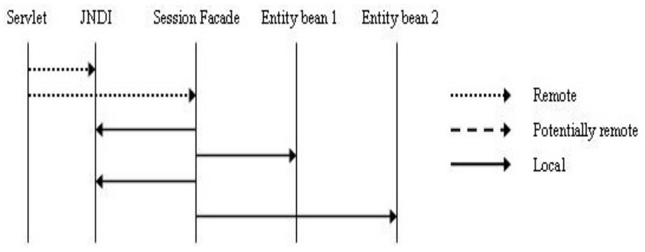

Figure 12 – Communications involved in a request with CMI

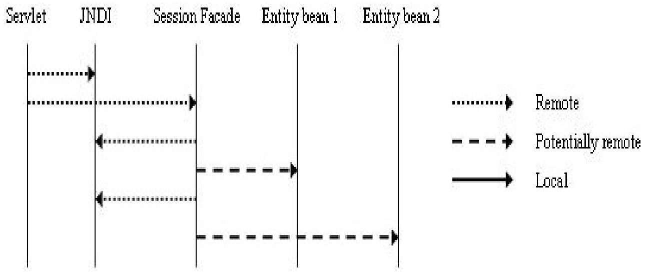

Figure 13 – Communications involved in request with partitioned EJB

6.3 Variant of the solution

This variant aims at reducing as much as possible remote communications. In this purpose, we replicated the JNDI service similarly to the CMI solution, as illustrated in Figure 14.

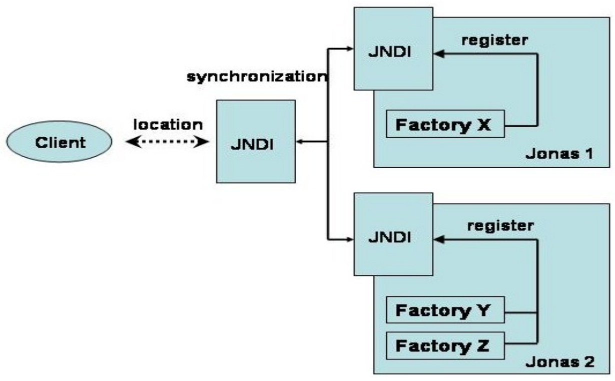

Figure 14 – Factory partitioning and JNDI service replication

Figure 15 reports the obtained performance results (with the same hardware configuration). The maximal throughput is about 16 requests per second for a number of 130 clients. These results are still poorer than CMI. The reason is that factory partitioning generate a large number of remote invocations between the two JOnAS servers, between remote beans. This overhead is not amortized by the advantage of having the cache enabled.

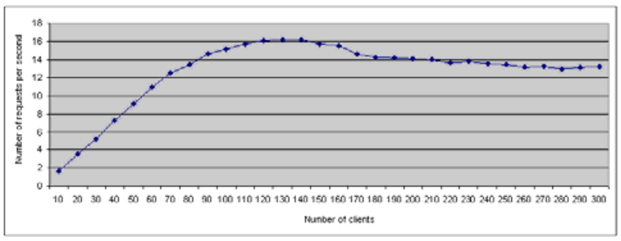

Figure 15 - Throughput with two EJB servers, partitioning, replicated JNDI service and cache enabled

A more accurate partitioning strategy (by bean and not by factory) would improve performance. However, we pursue in the next section with a hybrid solution (between partitioning and CMI) which appears to be much more suited.

7. Hybrid solution

7.1 Description

The principle of this solution is to distinguish read-only data from data which can be written. It requires knowledge of the data access pattern of the application. Read-only data are replicated in all the EJB servers while writable data are partitioned between the EJB servers. In both cases, caching can be enabled in all the EJB servers. This solution is hybrid between the two previous solutions. Figure 16 illustrates its architecture. Factories X,Y and Z manage writable beans and are partitioned, while factory A manages read-only beans and is replicated.

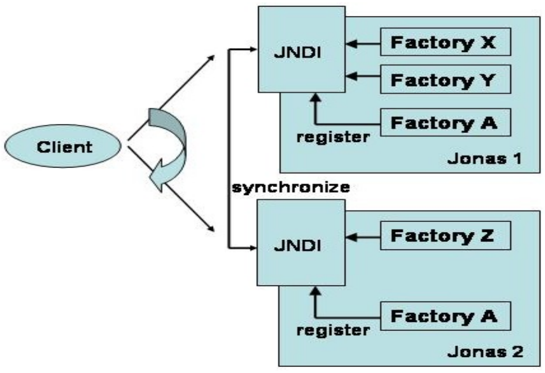

Figure 16 – Partitioning of writable data and replication of read-only data

The JNDI service returns a CMI stub for read-only bean factories and a RMI stub for writable bean factories. Therefore, invocations on read-only beans are distributed among replicas following CMI's load balancing algorithm (round-robin) and invocations on writable beans are routed towards the EJB server which hosts the invoked bean's factory. Caching is enabled in every EJB servers.

7.2 Performance

We replicated all the stateless session beans which don't raise any consistency problem. We analyzed the RUBiS application in order to identify beans that can be replicated. We found that entity beans Query, Category, Region, OldItem, Comment, Bid and BuyNow are read-only and that entity beans User, Item and IDManager are writable. Figure 17 reports the obtained results (with the same hardware configuration).

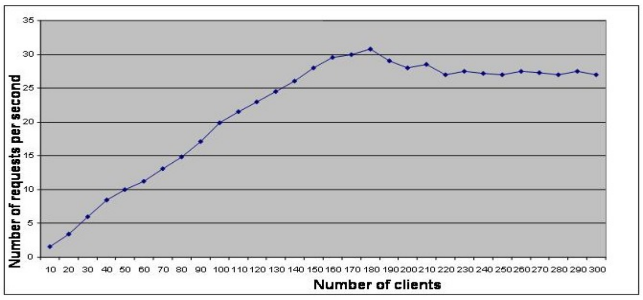

Figure 17 - Throughput with two EJB servers, CMI for read-only beans, partitioning for writable beans and cache enabled

We observe a maximal throughput of 30 requests per second for 180 clients. This hybrid solution improves performance by 20-30% compared to CMI. Its drawback is that it is not completely generic as it requires knowledge of the application.

8. Related work

Resource replication (at any level: disk, process, machine, etc.) has been much more studied in the purpose to provide fault tolerance than in the goal to provide scalability. However, fault tolerance and scalability are tightly coupled (especially in distributed systems) and often considered together, as illustrated by the definition of the *performability* metric [10]. Replication mechanisms introduced for fault tolerance often improve scalability (as a side effect). For instance, RAID-1 hard disks replication [12] tolerates a disk crash but also improve the performance of the overall persistent storage. This observation has motivated the C-JDBC project [2] whose objective is to provide scalability for relational database systems which enforcing their fault tolerance. C-JDBC exploits the principles of RAID to replicate a database on several machines and to maintain its consistency.

Resource replication allows obtaining scalability in two cases:

- The replicas are idempotent and the treatment of requests is not replicated (requests are distributed among replicas and their treatment always takes place on one server).
- The treatment of requests is replicated on several servers and the replicas are not necessarily idempotent.

For example, the *active* replication model and *primary-backup* replication model were designed to ensure fault tolerance, but they don't provide scalability [4].

Replication of stateless servers has been explored, especially for static web pages servers. Web servers can be replicated and requests distributed between replicas with a Round-Robin DNS or a L4 switch [7][13]. In the case of a statefull server, state consistency is the important issue; current solutions rely either on (i) broadcast of the state (or of updates) to the replicas [5][8] using group communication protocols (multicast IP, JGroup, etc.) or (ii) externalization of the state on a transactional support shared by all the replicas [6]; the choice of this transactional support is key to performance [3], but the most common choice (and inefficient) is to use a database system. Other solutions are considering replication in volatile memory in order to improve the management of this state [9].

Our hybrid solution is an alternative which consists in partitioning the state between a set of servers; therefore the state is distributed, but not replicated, which prevents the overhead due to consistency management. Fault tolerance is provided by the underlying transactional database system.

9. Conclusion

The J2EE specification allows designing multi-tiers application servers. A J2EE server is generally composed of four tiers, each being executed on a separate machine: a Web server tier, a Servlet server tier, an EJB server tier and a database server tier. The growth of the load that these servers (in terms of number of requests) may have to face raises the issue of the scalability of J2EE architectures.

The widely adopted approach consists in replicating the different tiers of the J2EE server, the unit of replication being the tier server as a whole. This approach has the advantage to manage replicas uniformly (the clones are not distinguished), which simplifies the routing of the requests (any replica can treat any request). However, the difficult issue raised by this approach is consistency management of state of a tier server when this state can be modified, which is the case for the EJB server. CMI is the solution of the JOnAS EJB server; CMI ensures consistency by synchronizing the replicas through the database tier (it forces propagation of accesses to the database), but it requires caching in JOnAS to be disabled.

In the context of a clusterized J2EE architecture, we compared the performance of a J2EE server with a single EJB server (with cache enabled) and with an EJB server replicated with CMI (with cache disabled). We observed and analyzed the benefits from using CMI. We then introduced a solution based on bean partitioning, which aims at improving the performance of CMI, by enabling EJB server replication and caching. The evaluation showed that this solution performs poorer than CMI, but our analysis allowed designing a hybrid solution between CMI and partitioning. In this hybrid solution, read-only beans are replicated with CMI (and can be cached) and writable beans are partitioned (a single copy is managed and can be cached). This solution improves the throughput of the J2EE server by 20-30% compared to CMI.

We are pursuing this work following two-directions:

- We wish to evaluate the strategies presented in this paper with a greater number of nodes and to study different partitioning policies, especially by beans instead of by factories.
- We are currently implementing facilities to dynamically adjust the number of server replicas according to the load that each tier has to face.

10. Bibliography

[1] E. Cecchet, J. Marguerite and W. Zwaenepoel. Performance and Scalability of EJB Applications. In Proceedings of OOPSLA, Seattle (USA), November 4th-8th, 2002.

[2] E. Cecchet, J. Marguerite and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In Proceedings of USENIX Annual Technical Conference, Freenix track, Boston, MA, USA, june 2004.

[3] G. Gama, K. Nagaraja, R. Bianchini, R. P. Martin, W. Meira Jr.and T. D. Nguyen. State Maintenance and Its Impact on the Performability of Multi-tiered Internet Services. In Proceedings of the 23rd Symposium on Reliable Distributed Systems (SRDS), Florianopolis, Brazil, October 2004

[4] R. Guerraoui, A. Schiper. Fault_Tolerance by Replication in Distributed Systems. Département d'Informatique Ecole Polytechnique Fédérale de Lausanne, 1996.

[5] Jakarta Tomcat Servlet Engine – http://jakarta.apache.org/tomcat/

[6] JOnAS Open Source EJB Server – http://www.objectweb.org

[7] E. Katz, M. Butler, and R. McGrath. A scalable http server : The ncsa prototype. Computer Networks and ISDN systems, 27:155–164, 1994.

[8] S. Labourey. Load Balancing and Failover in the JBoss Application Server », Sept 2003 http://www.clustercomputing.org/index.jsp?page=/content/tfcc-5-2-labourey.shtml

[9] B. C. Ling, Emre Kiciman, Armando Fox. Session State: Beyond Soft State. In Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004), San Francisco, CA, March 29-31, 2004

[10] J.F. Meyer. Performability Evaluation: Where It Is and What Lies Ahead. IEEE International Computer Performance and Dependability Symposium, April 1995.

[11] MySQL – http://www.mysql.com

[12] D. A. Patterson, G. Gibson and R.H. Katz. A Case for Redundant Arrays of In-expensive Disks (RAID). In Proceedings of the ACM SIGMOD International Conference onManagement of Data, Chicago, IL, USA, 109-116, 1988.

[13] S. Sudarshan, R. Piyush. Link level Load Balancing and Fault Tolerance in NetWare 6. NetWare Cool Solutions Article, March 2002.

[14] Sun Microsystems – Enterprise Java Beans Specifications – http://java.sun.com/j2ee/

[15] Sun Microsystems – Java Naming and Directory Interface (JNDI) – http://java.sun.com/products/jndi/

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