Database Access with EJB Application Servers Performance Study

Marcin Jarząb, Krzysztof Zieliński
Department of Computer Science
University of Mining and Metallurgy
Al. Mickiewicza 30, 30-059 Kraków, Poland

1. Introduction

Enterprise Java Beans (EJB) [6] is a server-side component architecture that simplifies the process of building enterprise-class distributed component applications in Java. This component technology originally proposed by SUN Microsystem is agreed upon by the industry, supports portability and rapid development of server side applications. EJB components (enterprise beans) are deployed within application servers (EJB containers), which provide the needed middleware.

EJB 2.0 specification defines three different kinds of enterprise beans: session beans which model business processes, entity beans which are the object that caches database information, and message-driven beans which are similar to session beans but could be called only by sending messages to those beans. Enterprise beans are not full-fledged remote object but their invocations are intercepted by the EJB container and then delegated to the bean instances. At the interception point the following major services provided by the EJB container are also available: transaction management, persistence, resource management and component life cycle, security, concurrency, and remote accessibility.

EJB containers are responsible for managing enterprise beans and interact with beans by calling management methods as required. A multi-tier architecture scalability is enhanced when EJB container intelligently manages the needed resources across a variety of deployed components. These resources could be threads, socket connections, database connections, and more. It means that the EJB container is responsible for coordinating the entire effort of resource management as well as managing the deployment beans' life cycle. The interface between enterprise beans and the EJB container is a fundamental point of this technology which is described in detail by the EJB specification [9]. Exact management schemes and their configuration attributes used by EJB container is application server implementation specific. This makes evaluation of Application Servers offered by different vendors rather difficult and opens an area for interesting performance study.

The paper deals with database access with EJB Application Servers performance study. Attention is paid to EJB container management schemes that support persistency and database operations. Their activity depends on many attributes set in configuration of EJB Application Server. These attributes control JDBC [18] operations or Container Managed Persistency (CMP) [9] and Transaction Service performed by the Application Server and could substantially influence the overall system performance. The number of these attributes, and inconsistence of their semantic definition across different EJB Application Servers implementation, make the choice of setting which provides the best performance for the given application a nontrivial task. In this paper two detailed aspects related to this problem gain a particular attention: (i) optimized entity EJB beans loading, and (ii) attributes settings for CMP. As alternative to entity beans Data Access Objects (DAO) have been considered and tested.

EJB based application performance is also very much depended on enterprise bean interfaces definition, their granularity, and data structure used for inter-communication. Thorough consideration of structuralization of the business processing and data access results in a proposal of design patterns [5]. The design patterns have been actively exploited in the presented study to eliminate an additional overhead introduced by the inefficient usage of the EJB technology.

The paper is structured as follows. In Section 2 patterns used for structuralization of database access via application server are shortly described. Next, tuning mechanisms applicable to most of EJB Application Servers such as control of transaction behaviour, tune thread count and some features proprietary for application servers vendors are discussed. Next, performance study methodology and scenarios are described. Performance test results of three most popular Application Server such as BEA Weblogic, JBOSS and Sun ONE are presented in Section 4. The obtained results are compared and summarized in Section 5. The paper is ended with conclusions.

2. Database Access Design Patterns

Building applications with the EJB technology in an efficient way requires very good understanding of the key characteristics of this middleware platform. The experience gained by application programmers in many areas of software engineering has been summarized as Design Patterns. Pattern is a solution to a recurring problem in the context. Once Pattern (solution) is developed from a recurring problem that can be reused many times without reinventing the solution again. Patterns are popularized by the classic book [1]. Specifically for EJB problems and solutions, we have now Core J2EE Patterns, Best Practices and Design Strategies defined by Sun Java Center [2] and EJB Design Patterns accessible from [7].

This section mainly focuses on performance improvement practices using Patterns in EJB. As many reports [11] referring to this issue exist we limited the presentation only to the selected solutions which we apply in our study. To understand the organization of these Patterns it is necessary to distinguish between session and entity beans. Session beans are business process objects which are relatively short-living component. Their lifetime is roughly equivalent to a session or lifetime of the client code that is calling the session bean. The two subtypes of session beans are stateful session beans and stateless session beans. A stateless bean is a bean that holds conversations that span a single method call. After each method call, the container may choose to destroy a stateless session bean, or recreate it, cleaning itself out of all information pertaining to past invocations. It also may choose to keep the instance around, reusing it for all clients who want to use the same session bean class. The exact algorithm is container specific. In fact , stateless session beans can be pooled, reused and swapped from one client to another client on each method call. This saves time of object instantiating and memory.

With stateful session beans, pooling is not as simple. When a client invokes a method on a bean, a client is starting conversation with the bean, and the conversational state stored in the bean must be available for the same client's next method request. Therefore, the container cannot easily pool beans since each bean is storing state on behalf of a particular client. But we still need to achieve the effect of pooling for stateful session beans to conserve resources and enhance the overall scalability of the system. EJB containers limit the number of stateful session beans instances in memory, by swapping out a stateful bean, saving its conversational state to a hard disk or other storage. This is called passivation. When the original client invokes a method , the passivated conversational state is swapped in to a bean. This is called activation. The bean that receives the activated state may not be the original bean instance, but that's all right. The container decides which beans to activate and which beans to passivate and it is specific to each container. Passivation may occur at any time, as long as a bean is not involved in a method call or transaction.

Entity beans are persistent objects which are constructed in memory from database data, and they can survive for long periods of time. This means if you update the in-memory entity bean instance, the data base should automatically be updated as well. Therefore there must be a mechanism to transfer information back and forth between Java object sand database. This data transfer is accomplished with two special methods that entity bean class must implement, called *ejbLoad* and *ejbStore*. These methods are called by the container when a bean instance needs to be refreshed depending on the current transactional state.

The EJB technology assumes that only a single thread can ever be running within a bean instance. To boost performance it is necessary to allow containers to instantiate multiple instances of the same entity bean class. This allows many clients to concurrently interact with separate instances, each representing the same underlying entity data. If many bean instances represent data via caching , we are dealing with multiple replicas cached in memory. Some of these replicas could become stale, representing data that is not current. To achieve entity bean instance cache consistency, each entity bean instance needs to be routinely synchronized with underlying storage. The container does this by calling the bean's *ejbLoad* and *ejbStore* methods. The frequency with which beans are synchronized is dictated by transactions, which give clients the illusion that they have exclusive access to the data. Similarly as session beans, entity beans instances are objects that may be pooled depending on the container's policy. It saves the resources and shortens the instantiating time. When an entity bean instance is passivated, it must not only release held resources, for example, the database connection but also save its state to the underlying storage by calling *ejbStore*. Similarly, when the entity instance is activated, it must not only acquire certain resources it needs but also load the most recent data from database.

Enterprise beans encapsulate business logic with business data and expose their interfaces with all the complexity of the distributed services to the client. This could create some problems when too many method invocations between client and server lead to network performance bottleneck and overhead of many simple transactions processing. This problem is easily solved. We simply use session beans as objects encapsulating all business logic which exposes a kind of *API* , which can be used by a client to perform a certain work. Session beans should be used as a fascade to encapsulate the complexity of interactions between the business objects participating in a workflow. The *Session Fascade* manages the business objects and provides a uniform coarse-grained service layer access used by clients reducing network overhead. It is also important in situation when entity beans are transactional components, which means that each method call may result in invoking a new transaction, which can produce decreasing of performance. It is also important to note that each transaction commit results in database synchronization performed by EJB container. This behaviour can be controlled by encapsulating method calls of entity beans inside the session beans, which act as a transactional "shell" for all transactions raised by entity beans, thus leading to better performance.

The *Session Fascade* is one of the most popular EJB design patterns, which helps to obtain proper partition business logic and at the same time minimizes dependencies between a client and a server and forcing to execute business transaction in one networked call and in one transaction.

Session Fascade pattern usage could results in reduction of remote calls. A pattern which addresses only data transfer reduction overhead is the *Value Object* pattern. *Value Object* encapsulates a set of attributes and provides *set/get* methods to access them. Value Objects are transported by value from the enterprise bean to the client component. When the client requests the enterprise bean for the business data this bean constructs the value object, populate it with the attribute values and pass it by value to the client. Client, who calls an enterprise bean, which uses a value object, makes only one remote call instead of numerous remote calls to get each attribute value in each call. The client receives *Value Object* and invokes locally *set/get* methods on this object for accessing attributes values. It is necessary to point out that the same pattern could be used to optimise access to data stored in database.

Another problem is that access to data varies depending on the data source . Access to persistent storage varies greatly depending on the type of storage (RDBMS, OODBMS, LDAP flat files, and so forth) and the vendor implementation. These data must be accessed and manipulated from business components such as enterprise beans and other which are responsible for persistence logic. These components require transparency to the actual persistent store or data source implementation to provide easy migration to different vendor products, different storage types, and different data source types. The solution is to use *Data Access Object(DAO)* design pattern which abstracts and encapsulates all access to the data source.

The DAO design pattern enables transparency between business components and Data Storage. It acts as a separate layer which can be changed easily in case application migrates to other database implementation. Because the Data Access Objects manages all the data access complexities, it simplifies the code in business components that use the data access objects. All implementation-related code (such as SQL statements) is coded in the DAO and not in the business object. This improves code readability and development productivity. Also one important thing should be emphasized at this point. DAO is not useful for CMP entity beans, because EJB container serves and implements all persistence logic.

In performance study reported in this paper DAO is used as replacement for entity beans, so this technology will be described in more details. The Data Access Object manages the connection with the data source and implements mechanism to store and retrieve data . The DAO pattern can be made highly flexible by adopting the Abstract Factory and the Factory Method patterns as shown in Fig.1. This strategy provides a DAO factory object that can construct various types of DAO factories, each factory supporting a different type of a persistent storage implementation. Once the DAO factory for a specific data store is obtained , it it's used to perform persistence logic. The class diagram shows DAO factory as a base class from which different DAO factories inherit and implement specific storage access mechanisms to different implementations (for example, RdbDAOFactory to access an RDBMS such as Oracle, XmlDAOFactory to access an XML repository, and so on). Then, use a specific DAO factory such as RdbDAOFactory to obtain specific DAOs that support the business objects (for example, DAO1, DAO2, and so forth).

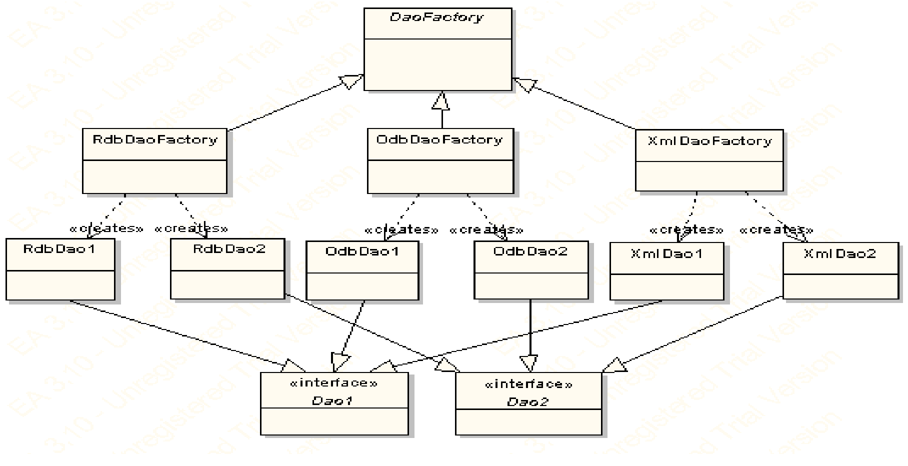

Figure 1. DAO Pattern concept

Session Fascade and *Value Object* should be treated as *EJB Layer Architectural Patterns* which should be taken into consideration during designing of EJB based application. There are also some tips which should be considered during implementation phase of enterprise beans and which in significant degree can tend to increase the performance. They are shortly described below:

- Serialization of *Value Objects* transferred between Remote Enterprise Beans should be considered and implemented in the most efficient way possible. To avoid sending the whole graph of objects a 'transient' key word should be used for the attributes that need not be sent over the network. Other solution is to implement this *value object* as multiple objects instead of coarse grained.

- References to Enterprise beans EJBHome object should be cached. There is already a pattern, which is called *Service Locator*, which is responsible for getting any objects from JNDI tree, and next put them into the cache. Next request for any of these objects wouldn't result in JNDI call, but the objects already stored in the cache will be returned. There is one very important detail which should be mentioned, namely what could happen if *Service Locator* is used in clustered environment and whether it is possible for cached *EJBHomes* to behave correctly, as for instance , the ability to load balancing requests and serving the requests in case the server fails or is restarted. The truth is that clustered servers always use *cluster aware home stubs* which implement all logic responsible for redirecting a client's request to appropriate cluster's node. The same answer must be applied to non-clustered environment where *home stubs* are also able to survive server restarts and crushes.

- Control transaction by avoiding transactions for non-transactional methods. If method calls must participate in transaction always appropriate transaction methods signatures should be declared to increase the performance of a transaction raised by this method call. The declarative transactions in EJB are at method level that means transaction starts (begins) when method starts and transaction ends (commits) when method ends. And also transaction propagates into the sub methods if the parent method uses these sub methods. For example, if you write a session bean method that calls four of the entity bean methods, transaction starts when the session method begins and transaction ends when that method ends, in between transaction propagates into four of the entity bean methods and gets back to session bean method. It works like a chain of transaction propagations. Declarative transactions have six transaction attributes: *Required*, *RequiredNew*, *Mandatory*, *Supports*, *NotSupported* and *Never*. So it is possible to control transaction to avoid unnecessary transaction propagation on every method. This can be done by dividing bean's methods into transactional methods and non-transactional methods and assigning transaction attributes to only transactional methods, assign 'NotSupported' or 'Never' to non-transactional methods to avoid transaction propagation. Please note that *'NotSupported'* or *'Never'* attributes cannot be used for entity beans because they need to involve in transaction to commit data, so these attributes can be used only for session bean's non-transactional methods. In this process we are controlling transaction propagation if any method uses other session beans but we have to be careful whether sub beans need a transaction or not. The transaction mechanism should span for minimum time possible because transaction locks the database data till it completes and it does not let other clients access these data.

- Use JDBC for reading**.** The most common use case encountered in distributed applications is the need to present a set of data resulted from certain search criteria, known as a *read-only use case*. When a client requests data for read-only purposes, solution, which uses entity beans, has some unnecessary overhead, which is often called *N+1* problem. In order to read N database rows when *entity beans* are used, one must first call *finder* method, which is one database call. Next for each acquired row, which is represented by entity bean *ejbLoad* method is called. Thus, a simple database query operation requires N+1 database calls when going through entity beans layer. Each such database call will temporarily lock a database connection from pool, open and close result sets and so on.
  Using JDBC for reading has some advantages. When we use JDBC queries to fetch data, queries are performed in one database call. All client data are acquired in single operation, which needs only one *connection* from pool, one *statement* and one *result set*. Comparing this behaviour to entity beans we can notice significant improvement of performance

- Some disadvantages of using entity beans were already mentioned when *JDBC for reading* use case was discussed. To solve some performance problems of entity beans, EJB specification offers option called *read-only entity beans.*
  The advantage of read-only entity beans is that their data can be cached in memory, booth in one server and many servers when dealing with clustering. Read only entity beans don't use expensive logic to keep the distributed caches coherent. Instead, the deployer specifies a timeout value and the entity bean's cached state is refreshed after the timeout has expired. Like any entity bean, the bean state is refreshed with the *ejbLoad* method call. When client invoke any method on *read-only entity bean* container ensures whether the associated data is older then timeout value. If this is a true synchronizing it's state is performed through *ejbLoad* method call. Because read-only entity beans don't participate in updating operations, *ejbStore* method is never called. One more thing is that read-only beans don't have to participate in transactions because their *ejbCreate* is never used. Read-only and read-write entities can live together when considering *read-mostly* design pattern. The concept of this pattern is the EJB optional deployment setting of read-only and to deploy the same bean code twice in the same application, once as read/write beans to support transactional behaviour, and once as read-only beans to enable rapid data access. In a read-mostly pattern, a read-only entity EJB retrieves bean data at intervals specified by the refresh-period deployment descriptor element specified in the descriptor file. A separate read-write entity EJB models the same data as the read-only EJB, and updates the data at required intervals. Main factor which should be considered when using

read-mostly pattern to reduce data consistency problem is to choose appropriate value of refresh interval. This should be set to the smallest timeframe that yields acceptable performance levels.

3. Tuning mechanisms in EJB servers during deployment phase

In this section the attributes of EJB Servers such as thread count, session and entity beans pools, and data source which are set in the deployment description are discussed.

- Tune thread count in EJB server. EJB server may have a facility to tune the number of simultaneous operations/threads (thread count) it can run at a time. If the default value of thread count provided by the server is less than the capability of the server, the clients requesting for an application may be put in a queue. Depending on resources and the capability of the server one should change the thread count to improve performance.

- Tune Session Beans. Optimisation practises discussed so far in Section 2 can be also applied to session enterprise beans , but there are also some details, which are specific to them. As we now we have two types of *session beans: stateless* and *statetful*. S*tateless beans* are not pined to any particular client, that means, they are returned to the pools when the business method has been executed. Every client who wants to perform any operation on this bean shares this pool. Of course in this pool there is only a limited number of beans, so if there are more requests than numbers existing in the pool these requests are queued. There is a possibility to specify minimum and maximum instances of session beans in vendor deployment descriptor. These values should be adjusted to number of clients who will perform any operation on that enterprise bean. If a session bean acts as a *Session Facade*, then *setSessionContext* should be used for setting references to *entity beans* EJBHome handles. Here should be also fetched other resources like session beans, data sources which can be used during life cycle of this bean.

- Tune Entity Beans. The same optimisation practises as for session beans pool can be also applied to entity beans pool. At this point one thing should be emphasized, namely entity beans are responsible for persistence, thus their behaviour is much more heavyweight than session beans. Activation and passivation during lifetime of entity beans are expensive. For every activation the Container calls *ejbLoad* to get latest data from the database and calls *ejbActivate* method. For every passivation the Container calls *ejbStore* to store data in the database and calls *ejbPassivate* method. Methods *ejbLoad* and *ejbStore* communicate with the database to synchronize the latest data. If the number of concurrent active clients (when the client calls business methods) is bigger than instance cache size then activation and passivation occur often thus effecting the performance. So in order to increase the performance, the optimal cache size must be set. The cache size must be equal to concurrent active clients accessing the bean. The instance cache size and pool size in entity beans are larger than session beans. The beans in the pool and the cache should accommodate the entity beans requirements like finder methods that return large number of records and populate the data in the beans. So we should be careful when we configure entity bean pool size and cache size.

- Use transacted *TxDataSource* instead of non-transacted *DataSource*. The main difference between *TxDataSource* and *DataSource* is ability to handle distributed transactions across multiple databases. Also the connections are handled differently. Non-transacted *DataSource* always grabs a connection from a pool with *autoCommit* flag set to true, which means that each update is immediately commited to DB. *TxDataSource* recognizes the fact that there is a transaction in progress and if the connection was requested for the first time it sets its *autoCommit* to false and associates it with the current transaction. This means that connection will not be returned to the pool until the moment transaction completes and all subsequent *DataSource.getConnection()*calls return the same connection.

The presented technical issues are very important for EJB Server activity related to *Container Managed Persistence* support. They are manifested as an extension to CMP known as optimised loading and commit options. Optimized loading option is to load the smallest amount of the data required to complete the transaction in the least number of queries. Optimized loading helps to avoid *N+1* problem when fetching data using entity beans. To use this option the application deployer must define *named-groups* for entity bean which contain only these bean data which are needed to perform transaction. These data include booth current bean fields among with relationships. This option is implemented in each application server evaluated in our tests, but naming convention in each of them is different. Commit options are also very important for loading process as they decide when an entity bean expires. EJB Specification 2.0 final Release specifies commit options A, B, and C defined as follows:

A. Container assumes that it is the sole user of database, therefore it can cache data of an entity bean between transactions, which can result in substantial performance gains.
B. Container assumes that there is more than one user of the database but keeps the context information about entities between transactions. This is the default commit option.
C. Container discards all entity bean context and data at the end of the transaction.

JBOSS implements also commit option D which is similar to A, except that the data only remain valid for a specified period of time.

4. Performance study methodology and scenarios

The goal of the performance study reported in this paper is a stress testing of a typical application implemented in J2EE environment. The stress testing is performed to ensure that the application scales appropriately handle the load for which it has been designed. An application called DSRG Training Activity Manager was used as a case study. This application supports educational activities like creating a new students' laboratory, assigning teachers to laboratories, creating new lessons, adding students, creating tests, etc. All information was stored in RDBMS database whose structure is depicted in Fig.2.

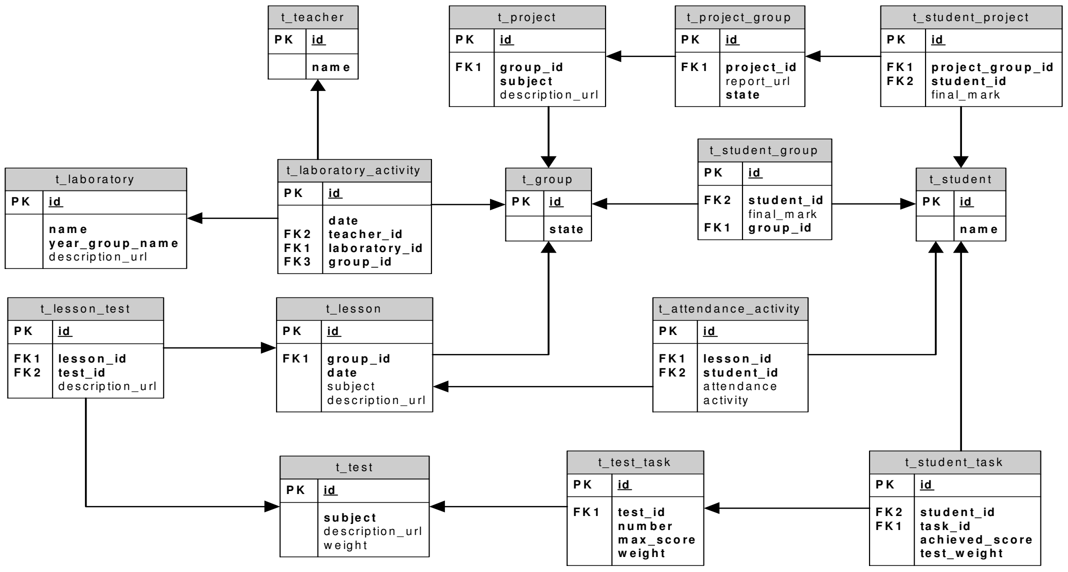

Figure 2. Database structure of DSRG Training Activity Manager

Three use cases which correspond to three typical operations on database were used for the testing purpose:

- Create Data - new lessons for a given activity group are created with attendance info and tests which can take place in this created lessons.
- Select Data - lessons info for a given activity group is fetched. This includes attendance info and tests scores which belongs to these lessons.
- Delete Data - delete info about lessons and test for a given activity group.

The investigated stress tests were oriented on database access performance study, so the data persistence mechanisms implementation was the most important . Two different approaches have been studied in this context:

- *Session Fascade* with DAO - DAO is responsible for implementing appropriate factory classes, which use JDBC API to implement access too database.
- *Session Fascade* with entity beans based on CMP 2.0 specification - all business logic responsible for persistence operations provided by EJB container which implements CMP 2.0 services.

These two approaches will be compared in more details in the following sections.

4.1.Session Fascade with DAO

With this approach, DAO objects are used to access data residing in RDBMS. The JDBC API enables standard access and manipulation of data in persistent storage, such as a relational database. Including the connectivity and data access code within session EJB component introduces a tight coupling between this component and the data source. Thus DAO design pattern is used to abstract and encapsulate all access to the data source which also manages all connections to store and retrieve all information. The architecture of DAO framework implemented for this purpose in our case study is shown at the diagram bellow:

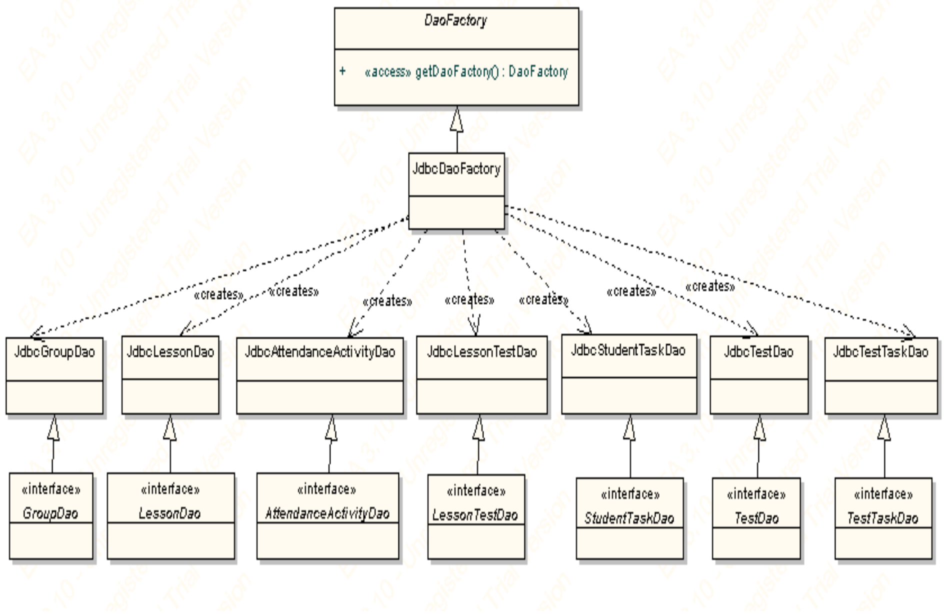

Figure 3. DAO framework class diagram

The important advantage of using direct JDBC is the ability to perform bulk SQL operations on the underlying relational database which helps to avoid multiple database calls, which helps significantly to achieve better performance. During this operations JDBC SQL parameterised query *java.sql.PreparedStatement* is used . Also transaction attribute *RequiresNew* were set for session bean methods which were exposed to the client.

4.2 Session Fascade with CMP 2.0

The EJB 2.0 model provides the improved support for container managed persistence for entity beans. EJB 2.0 supports better modelling capability for the bean provider in terms of what is called container managed relationships. This relationship can be implemented using local objects which offer lightweight access instead of using remote interfaces. Also all finder methods are implemented using EJB QL which ensures compatibility between all EJB containers. Fig.4 shows entities with relationships between them which represent data stored in RDBMS in our case study.

CMP 2.0 supports container managed relationships both in selecting and removing data. This enables to cascade deletes of all child data from any entity. One disadvantage of container manager relationships is the necessity to set explicitly the code relationships between entities when data are created. Transaction attributes for enterprise beans are set to *RequiresNew* for all client session bean methods and *Required* for entity beans.

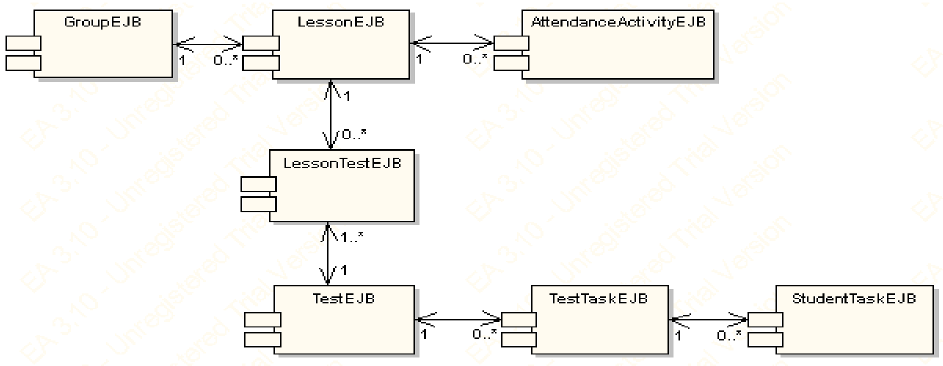

Figure 4. EJB entity beans participating in CMP 2.0 implementation

4.3. Load generation application

During the stress tests the presentation layer of the DSRG Training Activity Manager was replaced with the Grinder [15] load generation client application which was responsible for direct calls of session beans over IIOP-RMI, measuring performance and collecting all data. The Grinder is a pure Java load-testing framework that is freely available under a BSD style open-source license. Test client code is written in the form of Java "plug-ins". The grinder can simulate simultaneous clients accessing the application who can next perform any business transaction. It also records the time which elapsed from starting and finishing of a business transaction. The Grinder architecture, depicted in Fig.5, is quite sophisticated in spite of this it is very simple to use and to extend.

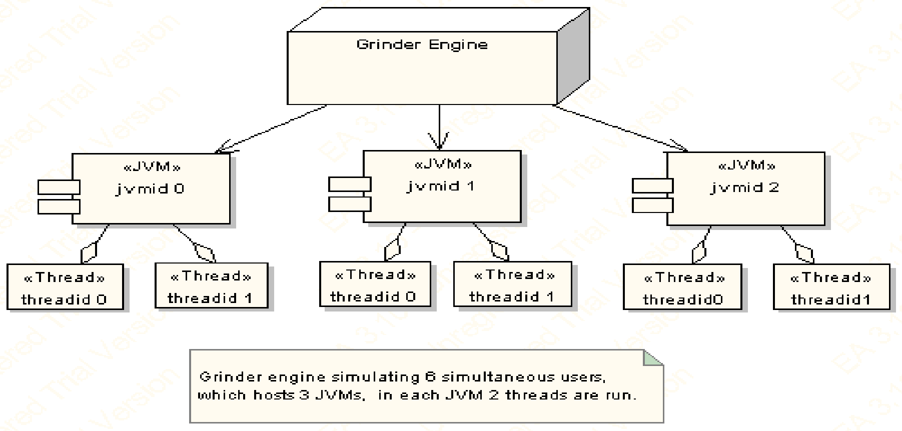

Figure. 5 Grinder engine architecture

The load applied in the performance study reported in this paper corresponds to the situation when a given number of clients is started at one by the Grinder. Each client performs the same business transaction which belongs to create, select of delete use case. After business transaction is finished successfully, the time, which elapsed from starting to ending of this transaction, is written to the Grinder's result-log file. When all transactions are finished the Grinder calculates *Average Response Time* (ART) as an average of the logged times.

5. Performance test results

For tests purposes only these J2EE application servers were considered which fully implements EJB 2.0 specification. Each application server was run with provided JVM, otherwise JDK 1.3 was used. This means that JDK 1.3 was used to evaluate test on Weblogic (provided with distribution) and JBOSS. In case of Sun ONE java runtime in version 1.4 were used which is also provided with the distribution.
As a database server Oracle9i was used with JDBC 4 thin driver. Also some standard configuration was modified to support the increased number of concurrent users. This includes increasing maximum number of processes *(default is 50)* to 500 and amount of open cursors *(default is 50)* also to 500. Oracle server was set up to work in a dedicated mode, which means that each physical connection (e.g. *JDBC Connection)* is served by one process.

Both the application server and database server were run on the same multi CPU machine *SUN Fire 6800* with 20 processors, 20 GB RAM, 120 GB hard drive and Sun Gigabit Ethernet interface and with Solaris 8 operating system. Grinder runtime was started on a separate machine *SUN Fire 3800* with 4 processors, 4 GB RAM, Sun Gigabit Ethernet interface and also with Solaris 8 operating system.

5.1 Test scenario

Use cases are performed in following order: *create, select, delete* with each implementation method i.e. *SF* with *DAO* and *SF* with *CMP 2.0* separately. Table 1 shows summary of how many rows are inserted in each table for given number of concurrent users in case of create use case. Delete use case is only performed for one user, who removes all entries created previously by all users. Select use case operates on number of entries created by 10 users.

Table 1. Number of rows inserted in each table by given number of concurrent users

Table name\Users	***10***	***50***	***100***	***150***	***200***	***250***	***300***	***400***	***500***	***600***	***700***	***800***	***900***	***1000***
t_lesson	*10*	*50*	*100*	*150*	*200*	*250*	*300*	*400*	*500*	*600*	*700*	*800*	*900*	*1000*
t_attendance_activity	*100*	*500*	*1000*	*1500*	*2000*	*2500*	*3000*	*4000*	*5000*	*6000*	*7000*	*8000*	*9000*	*10000*
t_lesson_test	*10*	*50*	*100*	*150*	*200*	*250*	*300*	*400*	*500*	*600*	*700*	*800*	*900*	*1000*
t_student_task	*400*	*2000*	*4000*	*6000*	*8000*	*10000*	*12000*	*16000*	*20000*	*24000*	*28000*	*32000*	*36000*	*40000*
t_test	*10*	*50*	*100*	*150*	*200*	*250*	*300*	*400*	*500*	*600*	*700*	*800*	*900*	*1000*
t_test_task	*40*	*200*	*400*	*600*	*800*	*1000*	*1200*	*1600*	*2000*	*2400*	*2800*	*3200*	*3600*	*4000*
Total:	***570***	***2850***	***5700***	***8550***	***11400***	***14250***	***17100***	***22800***	***28500***	***34200***	***39900***	***45600***	***51300***	***57000***

5.2. Application servers stress testing results

As it has been already mentioned three different application servers were evaluated for testing purposes, i.e. Weblogic 7.1, JBOSS 3.0.2 and Sun ONE 7.0.

Weblogic 7.1 contains some extra features which enhance performance of CMP entity beans. This includes *optimized loading (named groups and eager relationships fetching), db-is-shared (caching between transactions)* options and *read-only* entities.
Unfortunately in case of Weblogic 7.1 certain errors occurred upon performance of some tests. Utilizing CMP 2.0 implementation method for 100 concurrent users may be given as

example errors in selecting data use case. First error (Listing 1) was connected with EJB entity beans cache, which indicated that cache for entity beans was exceeded:

```
weblogic.ejb20.cache.CacheFullException
      at weblogic.ejb20.cache.EntityCache.put(EntityCache.java:363)
      at weblogic.ejb20.manager.DBManager.getReadyBean(DBManager.java:312)
```

Listing 1 .Weblogic exception which indicates out of size for entities

Default Weblogic size cache for entity beans is set to 100 and when this number is greater then exception shown above, it is thrown by EJB container. To avoid this error appropriate cache size *(max-beans-in-cache)* for entity beans in *weblogic-ejb-jar.xml* deployment descriptor must be set. Simple formula to set this size is as follows *execute_thread_count*number_of_data_returned.* After setting correct values for entity cache size, next error shown in Listing 2 occurred which is connected with Weblogic JTA timeout parameter which helps to avoid deadlocks.

```
java.sql.SQLException: The transaction is no longer active (status = Marked rollback. [Reason
=weblogic.transaction.internal.TimedOutException: Transaction timed out after 33 seconds
```

Listing 2 .Weblogic exception which indicates transaction timeout exception

Unfortunately, default value equal to 30 seconds was to small for these tests and 600 seconds was chosen. Last error was connected with J2EE application client, precisely with session beans stubs generated by Weblogic EJB compiler. These stubs have timeout (*RMI* timeout) value, which specifies the maximum time the client waits for response data from Weblogic server. The value by default is set to 240 seconds and if this is exceeded the exception is thrown and the client is unable to finish the business transaction. Unfortunately this error was the main drawback in these tests, especially when select data use case was performed.

JBOSS 3.0.2 offers some extensions to CMP: commit options (A, B, C, D), optimized loading (read-ahead), read-only entities. There is also one important feature, which is connected with mechanism of communication between JBOSS and remote clients. Namely for this purpose communication which relies on *Dynamic Proxies* architecture is used. The characteristic of this solution is that, stubs are not generated at the compilation time, but at the execution time using *Java Reflection API*. One of the drawbacks of JBOSS is the lack of possibility to set the executive thread pool count for the *EJB server* directly in JBOSS configuration files. The only way to control this, is to set up a HTTP server in the front-end of JBOSS. This cannot be applied during this test because direct access to EJB server through *JNDI* service is used.
Server option ***BlockingTimeoutMillis*** applies to *JDBC ConnectionPool* behaviour, namely this specifies how long a component has to wait for a desired connection in case there is no connection available. If this period is longer than this timeout value (by default this is 5 seconds), exception (Listing 3) is thrown.

```
ERROR - obtaining jdbc connection failed:No ManagedConnections Available!; - nested throwable
: (javax.resource.ResourceException: No ManagedConnections Available!)
```

Listing 3. JBOSS Exception which indicates timeout during fetching connection from pool

This timeout error was the main reason why some business transactions failed depending on the number of concurrent users. This situation is analogous when considering RMI timeout in case of Weblogic server.

Sun ONE 7.0 application server is an entirely new architecture which implements J2EE 1.3 and it's a part of Sun ONE platform. Sun ONE platform is Sun's standards-based software vision, architecture, platform, and expertise for building and deploying Services on Demand. It provides a highly scalable and robust foundation for traditional software applications as well as current Web-based applications, while laying the foundation for the next-generation distributed computing models such as Web services.
Some extensions to *CMP 2.0* include two *commit options B* and *C*, *optimized loading (named groups)* and *read-only* entities (works only for BMP). Interesting fact is that, *CMP 2.0* engine is based on *Java Data Objects* [19] technology which is treated as an alternative to entity beans. One of the drawbacks of current version of Sun ONE application server is the lack of clustering capabilities, but this will be implemented in future versions with commit options A and D. Thread pool size can be set separately for web server and ORB which is responsible for managing incoming client requests through RMI-IIOP.
During performance tests of Sun ONE we didn't face any serious errors like *timeouts*, concurrent access problems etc.

Table 2 collects runtime parameters for each application server under tests separately.

Table 2. Application servers parameters setting

Server	Java Runtime	Server specific settings
Weblogic 7.1	SUN JDK 1.3.1_03 Minimum heap size: 32 MB Maximum heap size: 200 MB	Execute thread count(thread pool size): 15 JDBC pool size: initial 15, maximum 20 Transacted data source: ***TxDataSource*** JTA timeout: 600 seconds
JBOSS 3.0.2	SUN JDK 1.3.1_03 Minimum heap size: 32 MB Maximum heap size: 200 MB	JDBC pool ***BlockingTimeoutMillis***: 480 seconds JDBC pool size: **Single instance**: initial 20, maximum 25 **Clustered instance**: initial 15, maximum 20 Non-transacted data source: ***DataSource*** JTA timeout: 600 seconds Default *HyperSonic service* shut downed.
SunOne 7.0	SUN JDK 1.4.0_02 Minimum heap size: 128 MB Maximum heap size: 256 MB	ORB thread pool size: 15 JDBC pool size: initial 15, maximum 20 Non-transacted data source: ***DataSource*** JTA timeout: 600 seconds

Fig.6 and 7 show result performance metrics adequately for *create use case* and *delete use case*. As we can notice *DAO* simply overkills *CMP 2.0* when used for deleting and removing and this situation takes place in all EJB containers. The only difference is that, we have various performance metrics, especially for *CMP 2.0* persistence container implemented in each EJB container. Implementation of *CMP 2.0* in Weblogic and Sun ONE has much better performance metrics when used for inserting and removing data then in JBOSS. The reason for this is probably that *Weblogic* and *Sun ONE* better supports bulk inserts and deletes, which increases performance of *Container Managed Persistence* during creating and removing data.

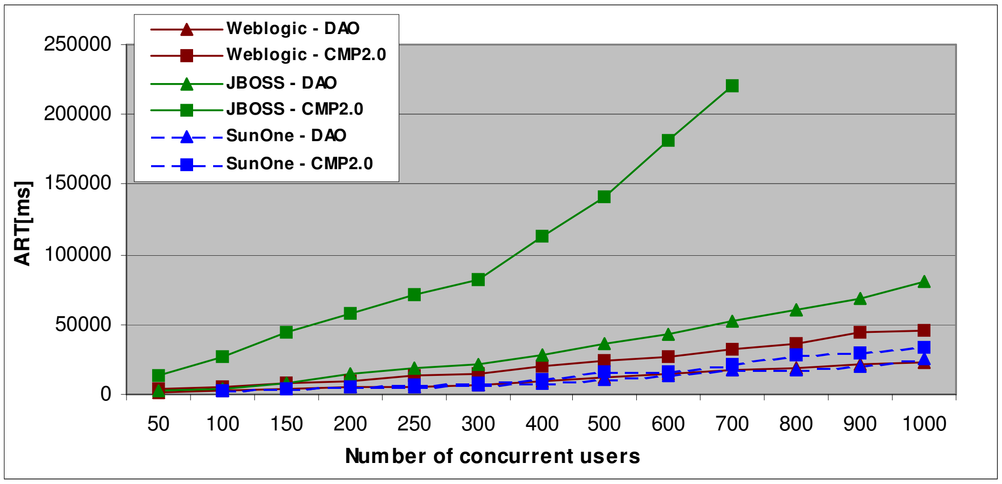

Figure 6. Create use case test results

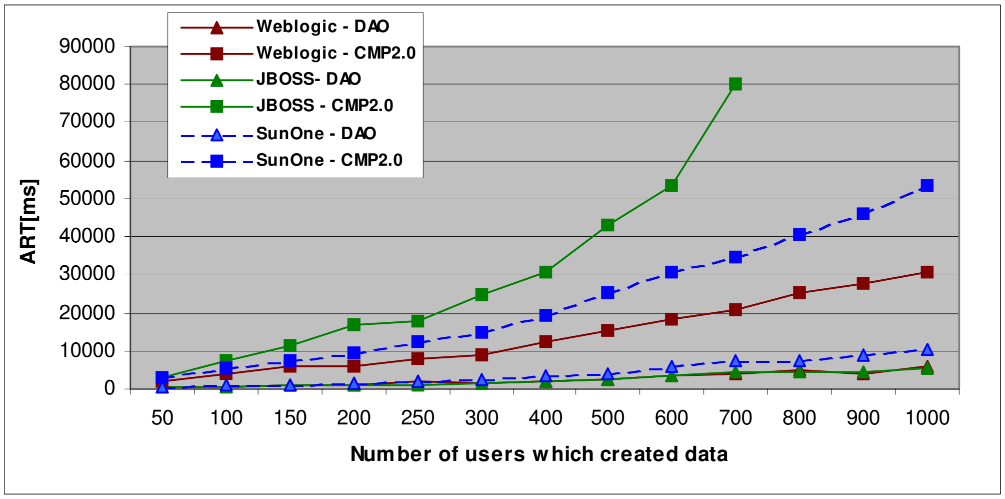

Figure 7. Delete use case test results

Performance metrics for *select use case* are shown in Fig.8. When analyzing *Weblogic* and *Sun ONE* response times we may come to the conclusion that *DAO* offers much better performance than *CMP 2.0.* Considering JBOSS results we have totally different conclusions; *DAO* rather doesn't have a significant impact on performance, it even scales worse at the application level. On the other hand, if we consider JBOSS CMP 2.0 extra options like *Commit A* or *read-only* entities results in performance are significant. The reason for which *Commit A* has better performance then read-only entities is probably *refresh-timeout* which indicates period for which cache must be refreshed according to database. For purposes of this tests refresh period was set to 150 seconds and there is a really big probability that entities were refreshed during tests.
The big surprise is a behavior of *read-ahead* (JBOSS) and *fetch-groups* (Weblogic, Sun ONE) options. Superiority over pure CMP can only be noticed only with *read-ahead,* but differences in response times are not so significant as may expected. When fetch groups are used the performance is even worst.

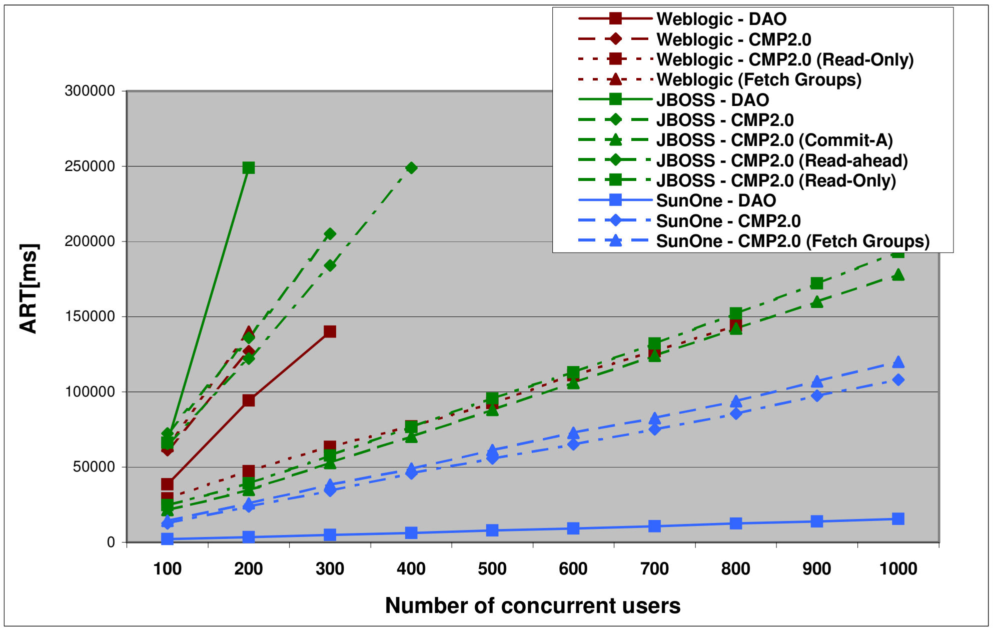

Figure 8. Select use case test results

5.3. Clustering

Scalability tests, shown in Fig.9 performed with JBOSS and Weblogic application servers shows that we can rely on clustering features offered by these application servers. The Sun ONE 7.0 does not support clustering at the moment. There is a significant improvement of performance when working on single instance and 2-node cluster, but differences between 2-node and 3-node are not so big according to response times.
The results of tests performed in clustered environment show that we were able to achieve better scalability; which means that there was a possibility to service more clients requests at the same time , and have the average response time decreased.

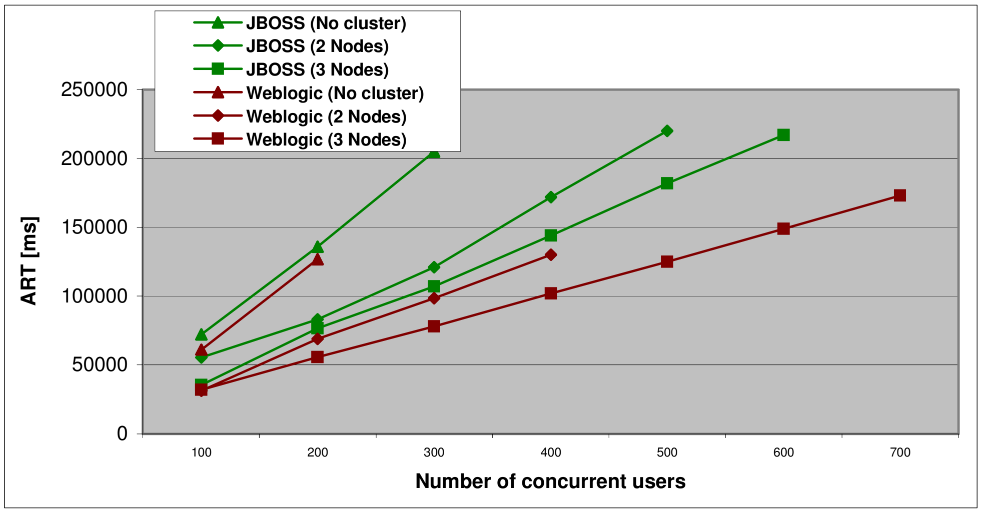

Figure 9. Select use case with CMP2.0 in a cluster

5.4. Application servers and JRE parameters setting

There is also possibility to play around some parameters which are specific both for a application server and java runtime. The scenario for application server may include thread poll size, entity cache size and other attributes which mostly are specific to the application server itself but may influence the performance of the whole application to a significant degree. The java runtime arguments may include some standard options e.g. *initial (Xms)* and *maximum (Xmx) heap* size and some other which are specific to the platform [16]. Setting the tests performed for Sun ONE 7.0 application server will be shortly discussed to illustrate the influence of these options. .
Baseline test concentrates on all parameters already mentioned i.e. thread pool size, entity cache initial and maximum heap size for JRE. Having analysed the results shown in Fig.10 we chose the following arguments mix for further tests : thread pool size equals to 20, entity cache with default values (initial and maximum size equals adequately to 32 and 64) and JRE initial and maximum heap size set to 256MB and 356MB. Next select use case with CMP 2.0 implementation method was performed. Comparisons metrics for two parameters mix are presented in Fig.11 and show that using arguments from baseline test offers average about 15% of improvement in performance.

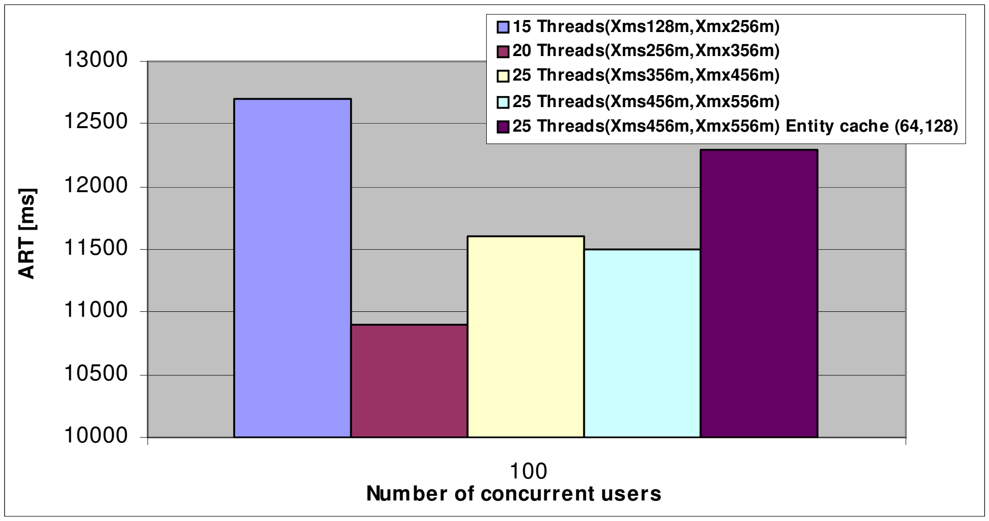

Figure 10. Baseline test for optimum thread pool and entity cache

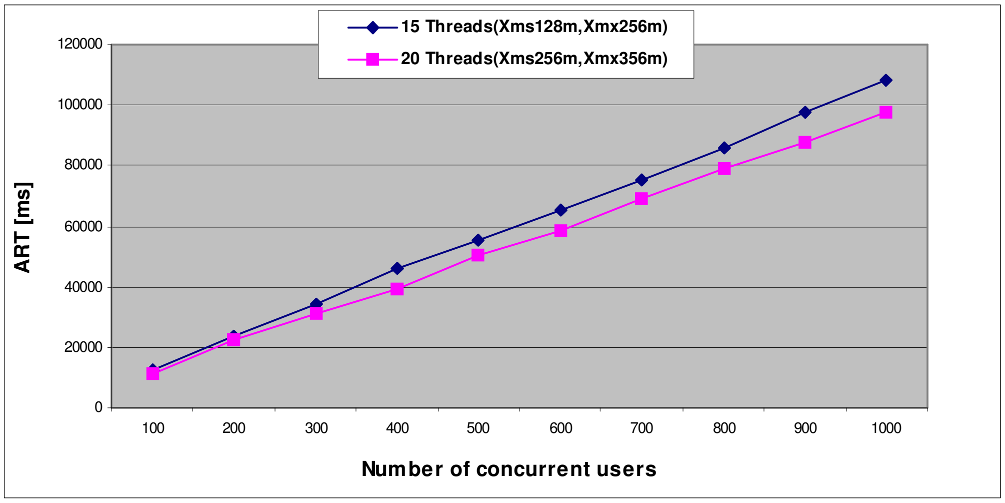

Figure 11. Select use case with appropriate thread pool and standard entity cache

5.5. Conclusions from tests

The obtained results proved that using DAO implementation method leads to better performance than CMP 2.0 when it was used to create or remove data. Analysis of performance metrics for the 'select' use case gives rather different view. When comparing DAO and CMP 2.0 with standard descriptor settings the first one gives significantly better performance when tested on Weblogic and Sun ONE. When analyzing some extra features related to CMP engine which are not strictly correlated with EJB specification like: *commit-A* or even *read-only* entities; using CMP 2.0 seems to be much more attractive than direct JDBC calls encapsulated in DAO. CMP 2.0 is much more flexible because of descriptors files which describe how each EJB component should be deployed into EJB container. Next thing which should be emphasized is that maintenance of DAO and JDBC code is much harder for developers and involve much more effort than persistence mechanisms implemented by the container. Nevertheless DAO offers the best performance in each tested use case when considering basic container options for CMP 2.0. This is specially seen in case of Sun ONE application server where DAO offers really good response times and is more ahead of it's competition i.e. Weblogic and JBOSS. Sun ONE behave very well also in create and delete use case with DAO and CMP 2.0 implementation method. The reason for this is probably a new architecture designed by Sun's engineers, but also that, we used JDK 1.4 for these tests, which introduces a lot of performance improvements [17] comparing to JDK 1.3. The big disappointment are performance results taken from JBOSS, because as we can notice it has the worst metrics from all the tested application servers.
As it has already been mentioned a lot of transactions weren't able to finish successfully because of timeouts on Weblogic and JBOSS especially during performing *select use case* both with DAO and CMP 2.0.

6. Summary

Performance testing of the EJB based application is rather a huge task and big challenge. There is a lot of factors at the application level and extensions offered by the application server. Appropriate adoption of these parameters involves holistic understanding of application and seems to be one of the most difficult part of the deployment phase.

The obtained results confirm that EJB performance is very sensitive to CMP Service attributes settings and could be easily destroyed even by wrong JRE parameters setting. In the case when an application requires higher scalability the alternative solutions to entity beans such as DAO should seriously considered.
This paper concentrates rather only on performance and scalability issues of middle tier where business logic of applications is implemented. We don't investigate these issues in the presentation and data tier. Good approach for the future is performance analyzing combined with optimizations practices, which can be applied to the data tier. This may include obtaining some statistics and performance reports after performing each use case with a given implementation method. Using these techniques will give a possibility to monitor an impact produced by each use case and implementation method. Further research can also be extended to the application server itself, not only to the performance tests but also to profiling. This may help estimate the impact on performance of each implementation method much better.

Bibliography

[1] A.Schaefer, *JBoss 3.0: Quick Start Guide*, JBoss Group, 2002

[2] D. Alur, J. Crupi, D. Malks, *Core J2EE Patterns - Best Practices and Design Strategies* Prentice Hall PTR / Sun Microsystems, 2001

[3] D. Bulka, *Java Performance and Scalability, Volume 1*, Addison Wesley Professional, 2000

[4] D. Sundstrom, *JBoss CMP*, JBoss Group, 2002

[5] E. Gamma, R. Helm, R. Johnson, J. Vissides, *Design Patterns*, Addison-Wesley Publishing Company, 1994

[6] E. Roman, S. Ambler, T. Jewell, *Mastering Enterprise JavaBeans, Second Edition*

[7] F. Marinescu, *EJB Design Patterns – Advanced Patterns, Processes, and Idioms* John Wiley & Sons, Inc., 2002

[8] J. Shirazi, *Java Performance Tuning*, O'Reilly, 2000

[9] L. G. DeMichiel, S. Krishnan, L. Ü. Yalçinalp, *Enterprise Java Beans specification version 2.0,* Sun Microsystems, 2001

[10] M. Girdley, R. Woollen, S.L. Emerson, *J2EE Applications and BEA WebLogic Server,* Prentice Hall PTR, 2001

[11] P. Gomez, P. Zadrozny, *Java 2 Enterprise Edition with BEA Weblogic Server*

[12] R. Monson-Haefel, *Enterprise JavaBeans, 3nd Edition,* O'Reilly, 2001

[13] R. Adalta, F. Arni, K. Gabhart, J. Griffen, M.B. Juric, A. Mulder, D. O'Conner, J. Lott, T. McAllister, A Mulder, N. Nagarjan, T. Osborne, P.G. Sarang, A. Tost, D. Young, Craig.A. Berry, *Professional EJB*, WROX Press Ltd.,2001

[14] S. Wilson, J. Kesselman, *Java Platform Performance, Strategies and Tactics*, Addison Wesley, 2000

[15] *Grinder Home Page* http://grinder.sourceforge.net

[16] *Java HotSpot VM Options* http://java.sun.com/docs/hotspot/VMOptions.html

[17] *Java 2 Platform Standard Edition, Performance and Scalability Guide* http://java.sun.com/j2se/1.4/performance.guide.html

[18] *JDBC Home Page http://java.sun.com/products/jdbc*

[19] *Java Data Objects* http://java.sun.com/products/jdo