TECHNIQUES TO IMPROVE SCALABILITY AND PERFORMANCE OF J2EE-BASED APPLICATIONS

MARCIN JARZĄB, JACEK KOSIŃSKI
AND KRZYSZTOF ZIELIŃSKI

*Department of Computer Science, AGH University of Science and Technology,
Al. Mickiewicza 30, 30-059 Cracow, Poland
{mj, jgk, kz}@agh.edu.pl*

(Received 6 June 2004; revised manuscript received 28 August 2004)

Abstract: This paper reports research on improvement techniques of scalability and performance of Java 2 Platform, Enterprise Edition, (J2EE) based applications. The study deals with operating systems and Java Virtual Machine (JVM) tuning, the setting of Enterprise Java Beans (EJB) Application Servers' configuration parameters and clustering methods. The theoretical principles of achieving high performance and scalability of J2EE Applications are considered. The experimental environment and scenarios are described. The experimental results of the considered techniques' evaluation are presented and analyzed.

Keywords: J2EE, EJB, performance, scalabilty, tuning, load-balancing

1. Introduction

The Java®2 Platform, Enterprise Edition, (J2EE®) [1] is the standard server-side environment for developing enterprise applications in the Java programming language. These applications are typically composed of several tiers that handle specific aspects, notably web modules (Servlets and Java Server Pages) for interaction and presentation, EJB modules for business logic, and resource adapter modules for accessing legacy applications. These modules are hosted in containers that interpose between the application modules and the available services such as 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. Exact management schemes and their configuration attributes used by an EJB container is specific to the implementation of application server. This makes evaluation of Application Servers offered by various vendors rather difficult and opens an area for an interesting performance study.

The J2EE architecture allows certain containers to be logically separated, *viz.* a Web container and an EJB container can be located in different JVM's, with communication through well-defined interfaces. In this way, the J2EE platform offers better scalability and more sophisticated deployment strategies. It is achieved by replication of application servers and transparent content switching of requests to support the suitable load-balancing between clustered instances. This technique could be further elaborated to provide fail-over when there is a necessity to assure fault-tolerant service operation for a huge number of simultaneously working users, whose number might reach a few thousands. The choice of a suitable load-balancing strategy and fail-over scheme is another interesting research area.

The already proposed solutions can be divided into two categories. One addresses the increase of application processing performance through proper selection of configuration parameters at the application server side as well as at the operating system side. The selection of proper application and operating system parameters is a complex and time consuming task, requiring substantial knowledge and experience. The other category assumes increasing the number of machines serving a given service. This mechanism is simpler in realization and guarantees the growth of performance through the purchase of additional servers and setting them up in the form of a farm. The performance benefits occur as processing simultaneous users' requests by distributing them among replicated nodes.

This paper summarizes research on improvement techniques of scalability and performance of J2EE-based applications performed during the last two years at the Department of Computer Science of AGH-UST. The goal of the paper is to practically demonstrate the influnce of operating systems and JVM tuning, the setting of EJB Application Servers' configuration parameters and clustering methods on system performance. It proposes a methodology of implementation and evaluation of the integration layer. A thorough consideration of structuralization of the business processing and data access results in a proposal of design patterns [2]. The design patterns have been used in the presented study to eliminate the additional overhead introduced by inefficient usage of the EJB technology.

The paper is structured as follows. In Section 2, J2EE patterns used to structure database access via an application server are shortly described. In Section 3, tuning mechanisms applicable to most of J2EE Application Servers, such as control of transaction behavior, tune thread count and some vendors' proprietary features of application servers, are discussed. In Section 4 we present clustering techniques applicable in J2EE environments. Section 5 contains the performance study's methodology and scenarios used for the tests. The performance test results of the three most popular Application Servers, BEA Weblogic [3], JBOSS [4] and Sun ONE [5], are presented in Section 6. The paper ends with conclusions.

2. J2EE Design Patterns

Building applications with the J2EE 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, popularized by the classic book [6]. Specifically for EJB problems and solutions, we now have Core J2EE Patterns, Best

Practices and Design Strategies defined by the Sun Java Center [7] and EJB Design Patterns [8].

This section focuses on performance improvement practices using patterns in EJB. As there are many reports [2, 9] referring to this issue, we have limited the presentation to selected solutions applied in our study. To understand the organization of these Patterns it is necessary to understand the EJB components' life cycle first.

2.1. EJB beans' life cycle-related techniques

While analyzing the EJB beans' life cycle it is useful to distinguish between session and entity beans. Session beans are business process objects which are relatively short-lived components. Their lifetime is roughly equivalent to a session or the lifetime of the client code that is calling the session bean. There are two subtypes of session beans: 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 may also 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 on each method call. This saves time of object instantiation and memory.

With stateful session beans, pooling is not so simple. When a client invokes a method on a bean, a client is starting a conversation with the bean, and the conversational state stored in the bean must be available for the same client's next method request. 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 the memory by swapping out a stateful bean, saving its conversational state to a hard disk or other storage. This is called passivation.

Entity beans are persistent objects which are constructed in memory from database data and can survive for long periods of time. This means that 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 objects and the database. This data transfer is accomplished with two special methods that the 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 the memory. Similar to session beans, entity beans' instances are objects that may be pooled depending on the container's policy. It saves resources and shortens the instantiating time.

2.2. EJB common Design Patterns

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 a client and a server lead to a network performance bottleneck and the overhead of many simple transactions being processed. To solve this problem, session beans should be used as a facade to encapsulate the complexity of interactions between the business objects participating in a business transaction. The *Session Facade* manages the business objects and provides uniform coarse-grained service layer access used by clients, reducing the network overhead. It is also important in the situation when entity beans are transactional components, which means that each method call may result in invoking a new transaction, possibly reducing the performance. This behavior 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 Facade is one of the most popular EJB Design Patterns, which helps to achieve a 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.

The use of the Session Facade pattern could result in a reduction of remote calls. A pattern which addresses only the data transfer reduction overhead is the Value Object pattern. A *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 business data from an enterprise bean, the bean constructs a value object, populates it with the attribute values and passes it by value to the client. The client who calls an enterprise bean that 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 a Value Object and locally invokes set/get methods on this object to access attribute values. It is necessary to point out that the same pattern could be used to optimize access to data stored in a 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, *etc.*) and the vendor implementation. These data must be accessed and manipulated from business components, such as enterprise beans, which are responsible for persistence logic. These components require transparency to the actual persistent store or data source implementation to enable easy migration to different vendor products, different storage types and different data source types. The solution is to use the 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 an application migrates to another database implementation. Because a Data Access Object manages all the data access complexities, it simplifies the code in business components that use the data access objects. All the 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. Another important point should be emphasized at this stage. DAO is not useful for Container Managed Persistence (CMP) entity beans, as an EJB container serves and implements all the persistence logic.

There are also some tips worth considering in the implementation phase of enterprise beans, which tend to significantly increase performance. They are briefly described below:

- Serialization of Value Objects transferred between Remote Enterprise Beans should be considered and implemented in the most efficient way possible.
- References to Enterprise beans' EJBHome object should be cached. There is already a pattern, called the Service Locator, which is responsible for getting any objects from the Java Naming and Directory Interface (JNDI) tree and putting them into the cache. The next request for any of these objects does not result in a JNDI call, but the objects already stored in the cache are returned.
- Transactions should be controlled by avoiding transactions for non-transactional methods. If method calls must participate in a transaction, appropriate transaction methods signatures should always be declared to increase the performance of a transaction raised by this method call.
- Use JDBC for reading. The most common use in distributed applications originates from the need to present a set of data resulting from certain search criteria, known as the read-only use case. When a client requests data for read-only purposes, a solution which uses entity beans has some unnecessary overhead, often called the $N+1$ problem. In order to read N database rows when entity beans are used, one must first call the finder method, which is one database call. The ejbLoad method is then called for each acquired row, represented by an entity bean. When we use JDBC queries to fetch data, queries are performed in one database call. Comparing this behavior to entity beans, we can notice a significant improvement in performance.
- Some disadvantages of using entity beans have already been mentioned above. To solve some performance problems of entity beans, the EJB specification offers an option of read-only entity beans.

The advantage of read-only entity beans is that their data can be cached in memory, on one or many servers, when dealing with clustering. Read-only entity beans do not use expensive logic to keep the distributed caches coherent. Instead, the deployer specifies a timeout value and the entity beans' cached state is refreshed after the timeout has expired. One more thing is that read-only beans do not have to participate in transactions.

Read-only and read-write entities can coexist in the *read-mostly* design pattern. The concept of this pattern is EJB optional deployment setting of read-only and deployment of the same bean code twice in the same application: as read/write beans to support transactional behavior and as read-only beans to enable rapid data access. In the 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 a read-only EJB and updates the data at required intervals. The main factor which should be considered when using the read-mostly pattern to reduce the data consistency problem is to choose an appropriate value of the refresh interval. It should be set at the smallest time-frame that yields acceptable performance levels.

3. Setting the operating system and application server parameters

Setting the operating system and application server parameters is most commonly referred to as tuning. The tuning that should be considered during the J2EE servers' installation and application deployment phase is related to:

- the operating system and TCP stack configuring;
- JVM parameter modification;
- J2EE servers' deployment setting.

In this section the main tuning techniques for each of these will be briefly described. This will illustrate the multidimensionality and complexity of J2EE environment setting.

3.1. The operating system and TCP stack

Tuning the operating system consists mainly in such selection of configurable system parameters that the usage of hardware resources is the most optimal. The selection of proper values strongly depends on the type of application whose performance is the most important. Typical parameters of special influence on the performance of operating systems are the size of swap space, its physical localization, and the manner of data access – the size of disk buffers or the settings related to communication with data storage devices such as DMA channel settings. The total performance may be improved by manipulating the configuration and parameters of the algorithm for scheduling processor tasks. This algorithm and its settings are particularly important for minimization of the so-called time of reaction, crucial for seriously overloaded systems. The Linux operating system enables modification of all these parameters, and through the availability of alternative implementations of ranking algorithms, matching the scheduler with a given application and the degree of system overload. In the case of the Linux operating system, performance can be improved by such kernel configuration that all unnecessary drivers are disabled. This significantly decreases kernel size and the size of the required memory.

Performance improvement of the TCP protocol communication subsystem means appropriate manipulations of time parameters that control the possibility of repeated device usage for the given socket or retransmission timers, connection establishment or dropping, *etc.* The present TCP stack implementations use efficient and effective methods of access to data that store information about open connections. These methods exhaust hash-tables. However, the parameters related to memory allocation for these hash-tables can be modified by such selection of parameters that their values would not create a bottleneck for TCP communications.

3.2. JVM tuning

When tuning JVM, the first parameter to be set is the overall heap space allocated and the sizes of the various generational areas [10]. Extensive testing has shown that the default parameters supplied by the Java server VM (using the server option) provide almost the best throughput, with the exception of a few parameters. To increase the predictability of garbage collection the settings for minimum (-`Xms`) and maximum (`-Xmx`) heap size should be set to the same value. The default value for both parameters is 64MB which may be too small in some cases, causing errors during runtime (`java.lang.OutOfMemoryException`). This factor should be considered especially with server applications, which in most cases have much greater memory footprint than common applications. The heap is also divided into areas, *viz.* young and old generation, which are supposed to hold objects of different ages. In some cases we may wish to customize the generation sizes, especially when the garbage collector becomes a bottleneck. Another important factor is the method of garbage collection (GC). Under heavy load, two factors have to be considered. First, if a large amount of heap space is allocated, garbage collection can take a long time, in the order of one, two or even three seconds. Second, hundreds or thousands of requests arrive per second and all of them have to be queued while the server performs GC. Often, there may be too many of them to be queued by the server process and the operating system's TCP stack. The TCP/IP connection resources get used up because the JVM is not processing any requests during a full GC, and all the queues, both of the application server and the operating system's TCP stack, become full of requests waiting to be processed. Apart from the default GC there are three additional collectors, *viz.* throughput, concurrent low pause and incremental. The throughput (`-XX:+UseParallelGC`) collector uses a parallel version of the young generation, while the old generation uses the default collector. This collector is similar to the default collector but with multiple threads making collections in the young generation. Another important GC is the concurrent low pause garbage collector (`-XX:+UseConcMarkSweepGC`). When using a concurrent GC scheme, extra CPU is consumed by the garbage collector all the time, which may lead to longer queue lengths and response times during the steady state, but results in much shorter collections times during a full GC.

3.3. J2EE servers' configuration during the deployment phase

In this section, the attributes of J2EE servers are discussed such as thread count, EJB container runtime parameters, *i.e.* session and entity bean pools, and parameters which affect Web server performance.

- Tune Web Server. If available, use a platform-optimized, native socket multiplexer to improve server performance. This extension is available in the Weblogic J2EE application server. Other parameters are AcceptBacklog (number of connections a server instance will accept before refusing additional requests, *i.e.* how many TCP connections can be buffered in a wait queue), KeepAlive and KeepAliveTimeout (specifies if HTTP persistent connections are supported and what is the timeout of each persistent connection).
- Tune thread count in the J2EE server. a J2EE 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 its capability, clients requesting an application may be put in a queue.

Depending on the resources and the server's capability, thread count should be adjusted to improve performance.

- Tune Session Beans. Optimization practices discussed in Section 2 can also be applied to session enterprise beans, but there are also some details which are specific to them. Every client who wants to perform an operation on a bean shares in the beans' pool. Of course, there is only a limited number of beans in this pool, so if there are more requests than beans in the pool, these requests are queued. There is a possibility to specify minimum and maximum instances of session beans in a vendor deployment descriptor. These values should be adjusted to the number of clients who will perform operations on that enterprise bean.
- Tune Entity Beans. The same optimization practices as for the session beans' pool can also be applied to the entity beans' pool. At this point it should be emphasized that entity beans are responsible for persistence, thus their behavior is much more heavyweight than that of session beans. Activation and passivation during the lifetime of entity beans are expensive. If the number of concurrent active clients (when clients call business methods) is greater than the instance cache size, activation and passivation occur, often affecting performance. Thus, in order to improve performance, an optimal cache size must be set. The cache size must be adequate to the number of concurrent active clients accessing the bean.

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 optimized loading and commit options. The optimized loading option is to load the smallest amount of data required to complete a transaction in the minimal number of queries. Optimized loading helps to avoid the $N+1$ problem when fetching data using entity beans. To use this option, the application deployer must define named-groups for each entity bean, containing only these bean data which are needed to perform the transaction. These data include the current bean fields together with relationships. This option is implemented in each application server evaluated in our tests, but the naming convention is different in each of them. Commit options are also very important for the 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 – a container assumes that it is the sole user of the database, therefore it can cache the data of an entity bean between transactions, which may result in substantial performance gains, B – a container assumes that there is more than one user of the database but keeps the context information about entities between transactions (the default commit option), C – a container discards all the entity bean context and data at the end of the transaction.

4. Building a scalable J2EE-based application using clustered solutions

A J2EE cluster consists of multiple server instances running simultaneously and working together to provide increased scalability and reliability. A cluster appears to clients to be a single server instance. The server instances that constitute a cluster can run on the same machine or be located on different machines. The cluster's capacity can be increased by adding additional server instances on an existing machine or adding machines to host the incremental server instances. The key clustering capabilities that enable scalability and high availability are (i) fail-over – when an application component doing a particular "job" becomes unavailable for any reason, a copy of the failed object finishes the job, and (ii) load-balancing – the distribution of jobs and associated communications across the computing and networking resources in a cluster's environment. For load balancing to occur there must be multiple copies of an object that can do a particular job, and information about the location and operational status of these objects must be available.

A clustered J2EE application is one that is available on multiple J2EE Server instances in a cluster. To simplify cluster administration, maintenance and troubleshooting, J2EE components should be deployed homogeneously to every server instance of the cluster. Moreover, applications may consist of different types of objects, including Enterprise Java Beans (EJB's), servlets and Java Server Pages (JSP's). Load balancing and fail-over for EJB objects is handled using replica-aware stubs, which can locate instances of the object throughout the cluster. Replica-aware stubs are created for EJB's as a result of the object compilation process during the deployment phase. In the case of EJB clustering we can only deal with proprietary solutions of a given J2EE application server. J2EE web containers for Servlets and JSP components support several techniques to achieve clustering: (i) Software, (ii) Hardware, or (iii) DNS load-balancers. For J2EE web containers, it is very important to ensure session persistence; the client remains associated with the server hosting the primary HTTP session object for the life of the session.

Software load-balancers provide the same functionality as hardware load-balancing, but in a software-only package. There are several products available on the market, both commercial (*e.g.* Resonate [11]) and free solutions based on Linux, like the Linux Virtual Server (LVS) [12].

There are also solutions which are built into J2EE application servers called Proxy Servers. Web servers built into J2EE application servers, or even Apache, Netscape and Microsoft Internet Information Services (IIS), can be used as proxy servers for routing requests to web containers. A proxy server is set up to redirect certain types of requests to the servers behind it. For example, a proxy server can be configured to handle requests for static HTML pages and redirect requests for Servlets and JSP's to a J2EE cluster behind the proxy.

LVS is an architecture which provides a framework to build highly scalable and highly available network services using a large cluster of commodity servers. The TCP/IP stack of the Linux kernel is extended to support IP load balancing techniques, which can make parallel services of different kinds of server clusters appear as a service at a single IP address. To use LVS the Linux kernel needs to be patched to support three IP load balancing techniques, based on LVS/NAT, LVS/Tunneling and LVS/DirectRouting. The core LVS functionality can only operate up to Layer 4 which means that it has no capabilities for inspecting HTTP headers, where cookies, *etc.* are stored, which helps to achieve persistence of the session mechanism. The solution of this problem is to use IP-based persistence. In this scenario, when a client first accesses the service, LinuxDirector will create a connection template between the given client and the selected server and then create an entry for the connection in the hash table. The template expires in configurable time, if an active connection exists. The connections for any port from the client will redirected to the server before the template expires. Although persistent ports may lead to a slight load imbalance among servers, it is a good solution for session persistence.

A hardware load balancer acts as a proxy to a cluster. The client connects to the load balancer and it routes the connection to one of the clustered servers behind it. Load-balancing hardware can track the "health" of each server and avoid sending requests to "unhealthy" servers; it can also incorporate load information in its load-balancing decisions. Two of the more apparent advantages of using load-balancing hardware are: (i) wider choice of load-balancing algorithms based on load, connections, or simple "round robin" logic, (ii) enabling Secure Socket Layer (SSL) acceleration dedicated hardware components that offloads SSL processing from the application server to the dedicated SSL accelerator. This can also significantly improve performance by decreasing the amount of time required to process secure transactions.

Some of the presented clustering techniques are evaluated in more detail in Section 6, where the results of performance tests are presented.

5. The performance study's methodology and scenarios

The goal of the performance study reported in this paper has been stress testing of typical applications implemented in the J2EE environment. The stress tests were performed to ensure that the applications were scaled appropriately to handle the load for which it had been designed. An application called the DSRG Training Activity Manager was used as a benchmark. This application supports educational activities like creating a new students' laboratory, assigning teachers to laboratories, creating new lessons, adding students, creating tests, *etc.* Our study is divided into two parts. In the first part, we investigate the influence of EJB Application Servers' configuration on performance when accessing databases. In the second part, we compare various load balancing techniques for J2EE web application servers' clustering.

5.1. Benchmarking EJB servers

Three use cases corresponding to three typical operations on a database were used for testing: (i) Create Data – in which new lessons for a given activity group are created, with attendance info and tests which can take place during these lessons, (ii) Select Data – in which lesson information for a given activity group is fetched, incl. attendance info and test scores for these lessons, (iii) Delete Data – in which information about lessons and test for a given activity group is deleted. The stress tests were oriented at a database access performance study, so the implementation of data persistence mechanisms was the most important. Two different approaches were studied in this context: (i) Session Facade (SF) with DAO and (ii) SF with entity beans based on the CMP 2.0 specification.

During the stress tests, the presentation layer of the DSRG[1] Training Activity Manager was replaced with the Grinder [13] load generation client application, which

1. Distributed Systems Research Group at AGH-UST

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

The use cases were performed in the following order: create, select, delete, separately with each implementation method, *i.e.* SF with DAO and SF with CMP 2.0. Table 1 shows a summary of how many rows were inserted in each table for a given number of concurrent users in the create use case. The delete case was only performed for one user who removed all entries previously created by all users. The select use case operated on a number of entries created by 10 users.

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

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

As has been mentioned above, three application servers were tested, *viz.* Weblogic 7.1, JBOSS 3.0.2 and Sun ONE 7.0.

Weblogic 7.1 contains some additional features which enhance the performance of CMP entity beans. These include optimized loading (named groups and eager relationships fetching), db-is-shared (caching between transactions) options, and read-only EJB's. The default Weblogic cache size for entity beans is set to 100 and when the number of instances is greater an exception is thrown which indicates that the cache is out of size. To avoid this error, an appropriate cache size must be set for entity beans in the deployment descriptor. A simple formula to set this size is as follows: execute_thread_count*number_of_data_returned. Another important factor is JTA timeout, which helps avoid deadlocks and must be increased in some scenarios.

JBOSS 3.0.2 offers some extensions to CMP: commit options (A, B, C, D), optimized loading (read-ahead), read-only EJB's. The JBOSS application server implements commit option D, which is similar to A except that the data remain valid only for a specified period of time. One of the drawbacks of JBOSS is the lack of possibility to set the executive thread pool count for the EJB server directly in the JBOSS configuration files. The only way to control this is to set up a HTTP server in the front-end of JBOSS. This method could not be applied during the test because direct access to the EJB server was used. BlockingTimeoutMillis server option is applicable to the JDBC Connection-Pool's behavior, namely it 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 the default timeout value (5 seconds), the exception is thrown by an EJB server.

The Sun ONE 7.0 application server is an entirely new architecture which implements J2EE 1.3 and is a part of the Sun ONE platform. The 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 and current Web-based applications, while laying the foundation for the next-generation distributed computing models such as Web services. During performance tests of Sun ONE we did not encounter any serious errors like timeouts, concurrent access problems, *etc.*

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

Table 2. EJB benchmark parameter settings

Element	Parameter values
Weblogic 7.1	JRE 1.3.1_03 Xms=32MB, Xmx=200MB 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	JRE 1.3.1_03 Xms=32MB, Xmx=200MB 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
SunONE 7.0	JRE 1.4.0_02 Xms=128MB, Xmx=256MB ORB thread pool size:15, JDBC pool size (initial 15, maximum 20), Non-transacted data source: DataSource, JTA timeout: 600 seconds

5.2. Benchmarking load balancer techniques for Web containers

This scenario evaluates three load balancing solutions: the Apache proxy, the J2EE server proxy and the Linux Virtual Server (LVS) using NAT. Compared with the previous scenario, the tested applications consisted of both EJB and JSP components. The use case was oriented at fetching data through SF mixed with DAO and the EJB-CMP approach. The goal of this test was to find the best load balancer solution offering the best throughput (number of requests per second) when accessing clustered nodes of a J2EE application server. Load was generated using the OpenSTA [14] software. Each virtual user (VU) performed an HTTP transaction which consisted of 21 requests (16 requests for JSP pages, which in turn spawn 5 requests for static content, gif's and style-sheets) for 20 iterations. The average thinking time between each request was 4.5 seconds, while thinking time oscillated between 1.5 and 8 seconds. The baseline test was focused on all the parameters mentioned in Section 3, *i.e.* Java runtime, Proxy servers and OS settings. Having analyzed the results, we chose the arguments presented in Table 3.

Table 3. Load-balancer benchmark parameter settings

Element	Parameter values
LVS	Slackware 9.1 with Linux kernel 2.4.21, gcc 3.2. All unused services removed from the kernel. LVS core engine compiled into the kernel, but additional features (like scheduling, routing policy) compiled as modules.
Apache proxy	Solaris 8 OS, opened file descriptors: 90000 Compilation settings: enable-non-portable-atomics, DFD_SIZE (opened descriptors) = 90000, worker MPM. Configuration settings: MaxClients:625, ThreadsPerChild:25, ServerLimit:25, ListenBacklog:100000, KeepAlive:Off.
Weblogic proxy	Execute Thread Count: 45, KeepAlive:Off, ListenBacklog:Unlimited, Native "IO" Performance Pack enabled, JRE: 1.4.02 version, -server, Xms/Xmx=512MB, default GC
Weblogic J2EE cluster	Cluster consists of 3 nodes. Node configuration: Execute Thread Count:15, KeepAlive:Off, ListenBacklog:Unlimited, Native "IO" Performance Pack enabled JRE: 1.4.02 version, -server, Xms/Xmx=256MB, default GC.

5.3. Hardware and software under test configurations

Only those J2EE application servers were considered which fully implement the EJB 2.0 specification. JBOSS 3.0.2, Weblogic 7.1, Weblogic 8.1 and Sun ONE 7.0 servers were evaluted for EJB database access performance.

Oracle9i with a JDBC 4 thin driver was used as a database server. Its standard configuration was modified to support the increased number of concurrent users. This included increasing the maximum number of processes and the amount of open cursors (both 50 by default) to 500. The Oracle server was set up to work in the dedicated mode, which means that each physical connection (*e.g.* an JDBC Connection) is served by one process.

We used SUN Fire 6800 with 24 processors, 24GB RAM and Solaris 8 OS. This server was divided into two separate domains: (i) Domain-1, 16 CPU's, 16GB RAM and 100 Mbps Ethernet, (ii) Domain-2, 8 CPU's, 8GB RAM and 100 Mbps Ethernet. The J2EE cluster was set up to host 3 nodes started on Domain-1. Apache and Weblogic proxies were installed on Domain-2. LVS was installed on a PC with Intel Pentium 4 – 2GHz, 512MB RAM, 100 Mbps Ethernet and Linux 2.4 OS. Load was generated by OpenSTA on 4 PC's with a twin configuration: Intel Celeron 2GHz, 512MB RAM, 100 Mbps Ethernet, Windows 2003.

6. Performance test results

The presented results concern the use cases and parameter settings described in the previous section. Only main results are described due to the limited length of the paper.

6.1. EJB servers' test results

Figure 1 depicts the result performance metrics adequately for the create and delete use cases. As we can notice, DAO simply overkills CMP 2.0 when used for deleting and removing, and this takes place in all EJB containers. The only difference

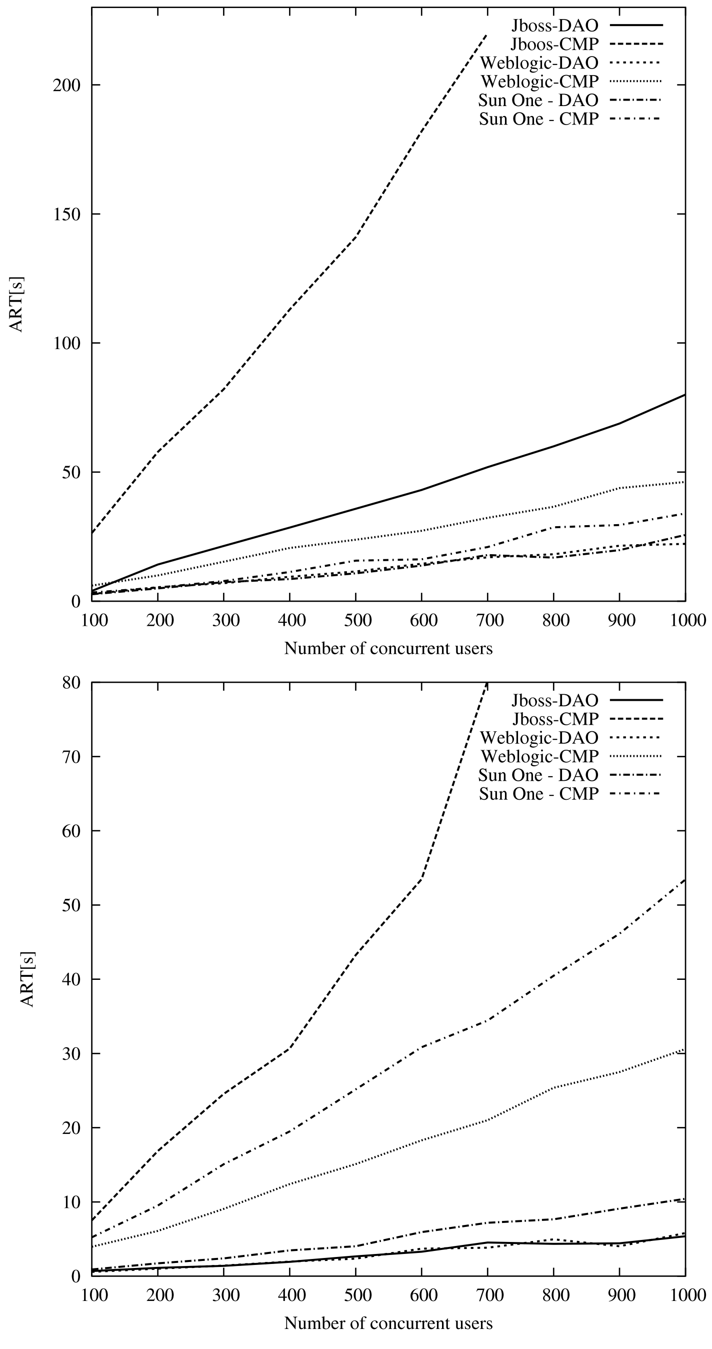

Figure 1. Test results of the create (upper panel) and delete (lower panel) use case

is that we have various performance metrics, especially for the 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 than its implementation in JBOSS. This may be due to Weblogic

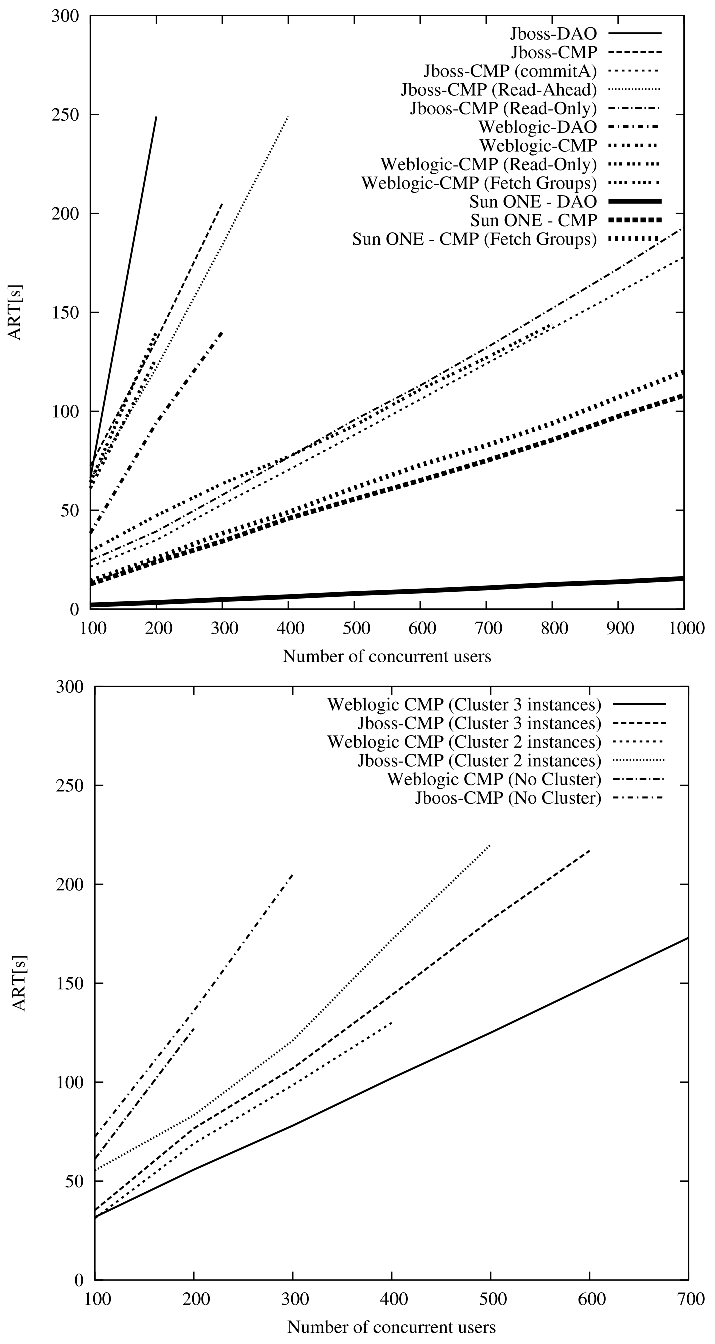

Figure 2. The upper panel depicts the test results for the select use case for a single EJB server; the lower panel depicts the same results with CMP 2.0 for clustered EJB servers

and Sun ONE better supporting bulk insertions and deletions, which improves the performance of Container Managed Persistence when creating and removing data.

Performance metrics for the select use case are shown in Figure 2 (upper panel). When analyzing the Weblogic and Sun ONE response times, we may come to the conclusion that DAO offers much better performance than CMP 2.0. Considering the JBOSS results we arrive at totally different conclusions; DAO has no significant impact on performance, it even scales worse at the application level. At the same time, if we consider the JBOSS CMP 2.0 additional options like commit A or read-only entities, results in performance are significant. That commit A has better performance than read-only entities is probably due to refresh-timeout, which indicates the period for which the cache must be refreshed according to the database. The refresh period was set to 150 seconds and it is very probable that entities were refreshed during the tests.

The behavior of read-ahead (JBOSS) and fetch-groups (Weblogic, Sun ONE) options has been a great surprise. Their superiority over pure CMP can be noticed only with read-ahead, but differences in response times are not as significant as may be expected. When fetch groups are used, their performance is even worse.

The obtained results have proved that using the DAO implementation method leads to better performance than CMP 2.0 when it is used to create or remove data. An analysis of the performance metrics for the "select" use case leads to a rather different view. When comparing DAO and CMP 2.0 with standard descriptor settings, the former offers significantly better performance when tested on Weblogic and Sun ONE. When analyzing some additional features of the CMP engine which are not strictly correlated with the EJB specification (*e.g.* 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 its descriptor files, which describe how each EJB component should be deployed into an EJB container. Another important point is that the maintenance of the DAO and JDBC code is much more difficult for developers and involves much more effort than the 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 especially evident in the case of the Sun ONE application server, where DAO offers very good response times and is further ahead of its competition, *i.e.* Weblogic and JBOSS. Sun ONE also behaves very well in the create and delete use cases with DAO and the CMP 2.0 implementation method. This may be due to the new architecture designed by Sun's engineers, but also to the fact that we used JDK 1.4 for the tests, which contains a lot of performance improvements [15] compared with JDK 1.3. The greatest disappointment are the performance results from JBOSS, as it has the worst metrics from all the tested application servers.

Scalability tests of EJB servers, shown in Figure 2 (lower panel), performed with JBOSS and Weblogic application servers demonstrate that we can rely on the clustering features offered by these application servers. There is a significant improvement in performance when working on a single instance and a 2-node cluster, but difference in response times between 2-node and 3-node is smaller. The results of tests performed in the clustered environment show that we were able to achieve better scalability, which means that it was possible to service more client requests at the same time and decrease the average response time.

6.2. Load balancers' test results

Figure 3 depicts results obtained for Apache, Weblogic Proxy and LVS. The superiority of the Weblogic solution over Apache which is a great surprise, considering

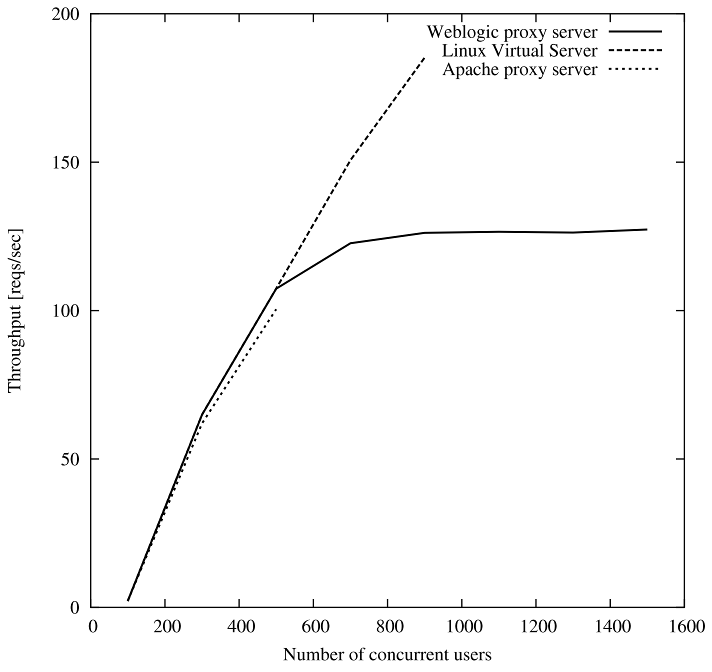

Figure 3. Load balancers' benchmark results

that Apache is implemented in C/C++. At the same time, modern JVM's offer Just-in-Time [16] optimization. Weblogic servers also use the Native-IO-Pack, which enhances I/O performance. In spite of these, Weblogic has scalability problems when 800 concurrent users access the application, and in this case the Weblogic proxy becomes a bottleneck for the whole J2EE clustered environment. Good scalability is offered by LVS until 900 concurrent users, when timeout errors occur. This is due to the mechanism for handling session persistence by LVS, which relies on IP-based persistence. In a test bed of 4 machines for load generation and 3 clustered nodes, IP-based persistence causes that one node in the cluster is overloaded as each request generated by 2 machines is redirected to the same node and the load is not proportionally distributed among the cluster's nodes.

7. Conclusions

Performance testing of J2EE-based applications is a huge task and a great challenge. There is a lot of factors at the application level and extensions offered by the application server. Appropriate adoption of these parameters involves a holistic understanding of the application and seems to be one of the most difficult parts of the deployment phase.

The obtained results confirm that EJB performance is very sensitive to CMP Service attribute settings and can be easily impaired, even by inadequate setting of the Java Runtime Environment (JRE) parameters. When an application requires greater scalability, solutions alternative to entity beans, such as DAO, should be seriously considered.

The test results of solutions that use load balance have revealed that simpler, low-level solutions of LVS software significantly surpass the software run in the user space in terms of performance. The frequent context switching between the kernel mode and the user space and data copying from sockets (absent from the LVS solution) are responsible for the decrease in performance of the solution using the Apache server.

References

[1] J2EE Platform Specification, http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

[2] Gomez P and Zadrozny P 2001 *Java 2 Enterprise Edition with BEA Weblogic Server*, Wrox Press Ltd.

[3] BEA Weblogic Server Overview, http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server

[4] JBoss Application Server Overview, http://jboss.org/overview/

[5] Sun ONE Application Server Overview, http://wwws.sun.com/software/products/appsrvr/home_appsrvr.html

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

[7] Alur D, Crupi J and Malks D 2003 *Core J2EE Patterns – Best Practices and Design Strategies Prentice Hall PTR*, Sun Microsystems

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

[9] Johnson R 2002 *Expert One-to-One J2EE Design and Development*, Wrox Press Ltd.

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

[11] Resonate Software Load Balancer, http://www.resonate.com/solutions/literature/data_sheet_cd.php

[12] Linux Virtual Server Project, http://www.linuxvirtualserver.org/

[13] Grinder Home Page, http://grinder.sourceforge.net/

[14] OpenSTA Home Page, http://www.opensta.org/

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

[16] Java JIT Compiler Overview, http://wwws.sun.com/software/solaris/jit