

EJB Performance Measurement Framework

Denis Gefter, M.S.
Robert Chun, Ph.D.

Computer Science Department
San Jose State University
San Jose, California 95192
dgefter@yahoo.com

ABSTRACT

This paper addresses challenges involved with measuring the performance of an Enterprise Java Bean (EJB) based system in a production environment. These challenges include specific requirements for the measuring system to be non-intrusive, have low-overhead, and to be able to provide monitoring and visibility to all the business objects in the production system. A prototype EJB Performance Measurement Framework capable of meeting these requirements is described. The proposed measurement framework also has an ability to combine JDBC monitoring with performance statistics measurements thus giving a broad overview of the production system's performance.

Keywords

Java, EJB, Performance, Profiler.

1. INTRODUCTION

Some general concepts appear repeatedly when assessing the performance of almost any Java enterprise development. The most important ones are: the concept of frequent measurement, and the concept of measurement in various configurations. Programmers are not necessarily good at guessing the source of performance problems. It is not uncommon to spend a lot of time tuning a module that affects only a small percentage of an application's total runtime and can only yield a very modest improvement. The problem is even more aggravated in systems using Enterprise Java Beans since they add additional layers of indirections. These indirections arise from application server services, such as those in application transaction management, application security, entity beans and failover. Other sources are design patterns like a session facade that groups application logic methods together.

So the role of performance measurement tools becomes even more important than before. Instead of guessing, the programmers routinely use various performance measurement tools such as profiling or time-stamped messages to show where the hot spots are located. While it is possible to simulate some load on the system in a laboratory environment, it is very likely that the behavior of the application in a production environment will be different. The main difficulty is being able to precisely measure and administer the workload. So it becomes important to collect performance details both during testing and deployment phases.

In tuning a program for better performance, there are a number of parameters that can be helpful to measure such as methods that are called, method runtime, average execution time, peak execution time, memory usage, throughput, database interaction and object creations. Several commercial profiling tools are available that can be helpful in measuring many of those quantities; however, most of them have a number of drawbacks.

The main one is the large amount of overhead introduced by a typical profiling tool. While it is totally appropriate to run a profiler in the laboratory environment, it becomes impractical to use one in a production setting.

Thus, there is often a need to collect performance and debugging information from a system after it has been deployed. In a typical scenario, performance details are collected during the testing phase of development. Once the system is ready to ship, the performance measurement code that was inserted during testing is removed. This invasive methodology creates several issues, such as constant code modifications, and potentially leaving behind debug-time code if measurement is still required in the production time system. So, it is important to create a performance monitoring system where changes in an application's source code are not required for measurement. Such a non-intrusive system would facilitate the important capabilities of being able to monitor the application in a production environment, tuning its parameters, and identifying potential bottlenecks.

So the purpose of this research is to present a performance monitor that provides the developers with detailed performance information about application objects and the events triggered by application methods in the system. This paper briefly explains the nature of EJB architecture and the challenges that are encountered in measuring the performance of an EJB based system. In the next section, a description of the relevant research previously performed in the area of EJB performance measurement is provided. Then, the system architecture and design details of the EJB Performance Measurement framework are discussed. Finally, performance test results are provided.

2. EJB ARCHITECTURE BACKGROUND

“An Enterprise Java Bean is a server-side software component that can be deployed in a distributed multitier environment. An enterprise bean can be composed of one or more Java objects because a component may be more than just a simple object. Regardless of an enterprise bean's composition, the clients of the bean deal with a single exposed component interface.”[18] The EJB specification defines three kinds of enterprise beans:

1. Session Beans that model business processes. Their APIs are typically designed to be used by clients and expose a rich functionality often presenting a “session facade” to the client.
2. Entity Beans that model business data. They are Java objects that cache database information. Session beans typically harness entity beans to achieve business goals.
3. Message-driven beans that are similar to session beans in that they are actions. The difference is that it is possible to call message-driven beans only by sending messages to those beans.

Figure 1 shows a high-level view of a simple EJB application. The client application is a servlet that receives data from a browser, and sends data to a browser as well. Also, the servlet interacts with an entity bean and a session bean through their home and remote interfaces.

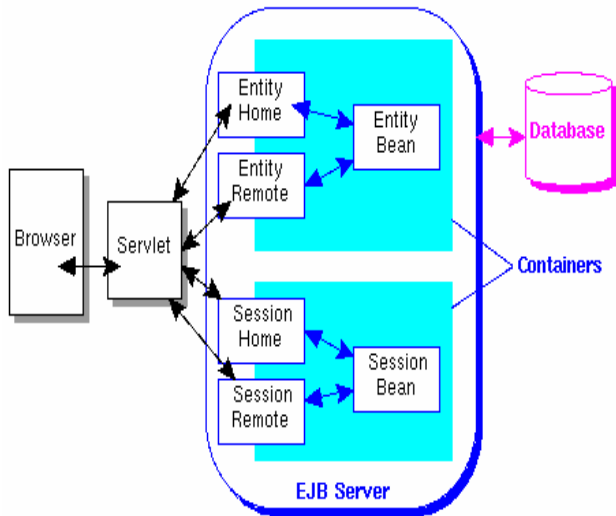


Figure 1: EJB Application

Distributed objects is one of the fundamental technologies that EJB components are based on. These objects can be called anywhere from the network. Figure 2 shows the communication process of a client with a distributed object. The process involves three major components:

1. The stub is a component that manages network communication over the network. In this manner the client is shielded from managing the sockets and serializing the parameters. The stub also implements a “remote interface” which exposes the methods that the distributed object wants the client to be aware of.
2. The stub calls a skeleton which abstracts the network communication details from a distributed object. The skeleton also is responsible for transferring the parameters from their network representation to a Java paradigm.
3. Once the distributed object receives the call and performs its function, it returns control over to the skeleton which calls the stub and, finally returns control to the client.

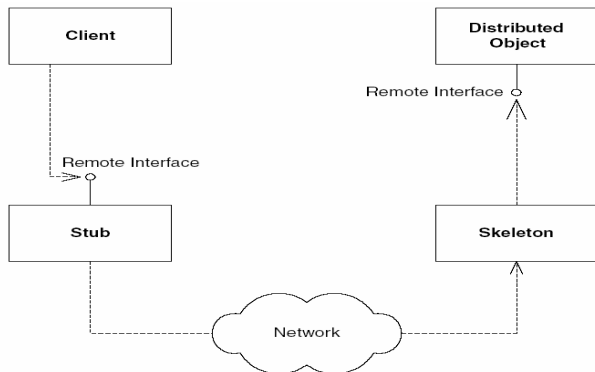


Figure 2: Client and distributed object

Distributed objects are implemented by OMG’s CORBA or Sun Java’s RMI-IIOP. A very important aspect of the EJB model is the manner in which any EJB client communicates with enterprise beans that reside in an application server. The client could be a servlet or JSP that resides inside a servlet container or a Swing application. Because of the transparent nature of the EJB architecture EJB uses naming and directory services. Naming and directory services are products that store and look up resources across a network. A typical API used to access naming and directory services is the Java Naming and Directory Interface (JNDI). In EJB, JNDI is used to lookup home objects.

So, when a client needs access to an enterprise bean, the following steps are performed:

1. The client obtains a reference to the Home Object of the deployed bean as specified by the deployment descriptor by using the Java Naming and Directory Interface (JNDI). Once the home reference is obtained, it is possible to either create or find an EJB Object (or collection of them) to access the bean business methods.
2. The EJB Home object creates a new EJB Object for the required bean. This object is not the actual bean but a server-generated object that acts as a proxy between the client and the enterprise bean class. This proxy has all the public methods of the enterprise bean and delegates the requests to the actual enterprise beans.
3. After the client receives a reference to the newly created EJB proxy, it can begin using the business methods of the bean. Even though the reference to the proxy appears to be local, all the methods are executed on the server and sent across the network.
4. The EJB Container intercepts all the calls to the actual bean and provides a variety of services to the bean implementation. The most common services include transactions, security, and resource pooling.

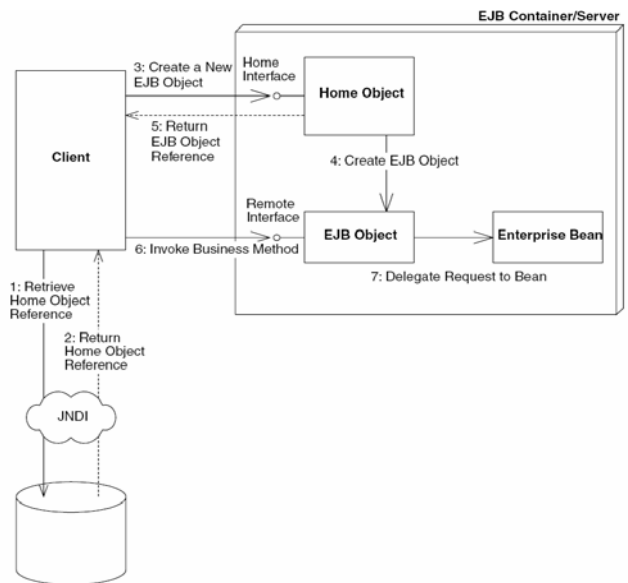


Figure 3: EJB Container Interaction

One of the shortcomings of EJB 1.0 and 1.1 was the high amount of overhead associated with calling an EJB Object. The overhead involved marshalling and de-marshaling parameters over the network, creating network connections and then repeating the process when an EJB object returned control to the client. In EJB 2.0 it is possible to call enterprise beans using their local objects that implement local interfaces rather than EJB Objects that implement remote interfaces. The communication process is greatly simplified for local objects:

1. Invoke local object from the client.
2. Local object manages connections, transactions, security, and other services.
3. Once the bean instance is finished with its work, control is returned to the client.

So, all the overhead work of marshaling/de-marshaling parameters, calling the stub, the skeleton and network is gone. However, the local interfaces have some significant limitations. One limitation is that local interfaces can only be called by the process in the same JVM. This means that the client for those local interfaces has to reside in the same application server as the local bean. It should also be noted that local interfaces marshal parameters by references (as opposed to remote interfaces that marshal them by value).

3. RELATED WORK

There are a number of tools and projects related to the performance monitoring of distributed systems. Below is a brief description of certain research projects in the area of Java enterprise performance monitoring and their relationship to the proposed system.

The Wabash tool [8] is designed for deployment testing and observing remote CORBA based systems. Since components are remote they get divided into various geographical regions. The Wabash tool features an interceptor that gets created in the address space of each server object and intercepts all incoming and outgoing requests to the server. The system has features of non-intrusive performance instrumentation. The system also uses the CORBA interceptor facility provided by Iona's OrbixWeb or Visibroker to catch incoming CORBA calls and collect server specific information.

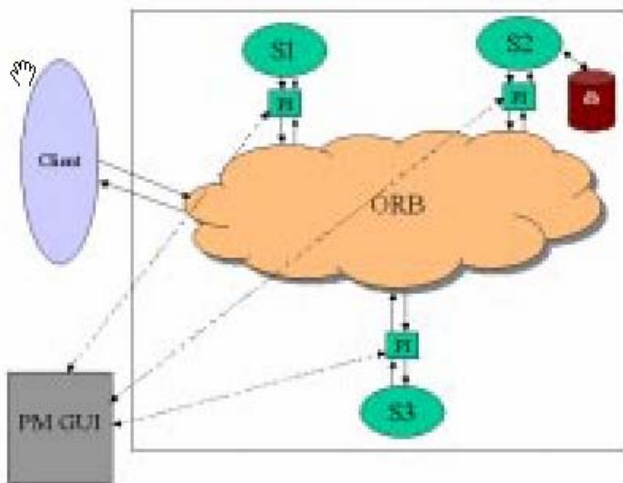


Figure 4: Wabash Architecture

The manager component interacts with the interceptor, and operates with the run-time store to save performance statistics. The Performance Instrumentation Agent that is part of the manager communicates with the GUI to retrieve information from the Stats module. The Wabash architecture has a certain similarity to the proposed framework in that it is non-intrusive as shown in Figure 4. However, it uses geographical information for monitoring components and that approach is not applicable in EJB environments where an application server controls the distribution of components.

The authors of "New Methods for Performance Monitoring of J2EE Application Servers" [14] present an interesting approach to monitoring existing Enterprise Java Beans applications. The authors explore the possibility of inserting the listener between the EJB object and the bean instance. This method does not require knowledge of the application server itself, but rather uses a "spy" approach where the monitoring application is deployed inside the application server itself. To accomplish that, the installation module inspects all the components of the application and creates a corresponding proxy for each and every one of them. So for each real enterprise bean a corresponding proxy bean will be created. Thus, all the calls to the real beans will be routed to the proxy beans instead, and all the clients that require API from original beans will be interacting with proxy beans instead. Since these proxy beans have the same API as the original beans, they will intercept each call invocation, notify the Monitor, and delegate the call to the original bean. Since the proxy beans have to expose the same set of interfaces as the original enterprise beans, the following sequence of steps must be performed:

1. Obtain and analyze original deployment descriptor XML files. These descriptors feature requests for the various services provided by the application server to the bean, such as transactions, security, persistence, etc. Descriptors also provide security information regarding authorized bean users and their respective roles.
2. By examining these descriptors and API that are required, the corresponding proxy bean is generated for each real bean. The generated proxy bean has the same name as the original bean.
3. Original beans are provided with the different alias that is known to the generated proxy beans.
4. All the beans (generated and original beans) are deployed to the application server.

The system is similar to the proposed tool in that it is non-intrusive and has little impact on run-time performance of the target application. However, it requires the generation of proxy beans and does not provide JDBC runtime information.

"JProfiler" [15] is a JVM profiler. It provides the following features: memory profiling, CPU profiling, and thread profiling. JProfiler uses a native interface to the JVM (the JVMPI) to get profiling information from a running Java application. The native library is loaded and initialized by adding a special directive to the command line (-Xrunjprofiler). As an application runs, the native library receives various types of events from which it builds its profiling database. JProfiler's GUI front-end connects to the profiled application through a socket. Each and every method call that is made in the JVM is reported to the JProfiler agent library for separate measurement. To speed the measurement, JProfiler uses a concept of "dynamic instrumentation". Dynamic

instrumentation works by instrumenting Java byte code on the fly as the JVM loads classes from external storage. The raw class files are intercepted by JProfiler's native agent library before the JVM parses them. Appropriate byte code instructions for profiling are added at the beginning and the end of each method as well as around calls into filtered packages. The byte code modification is done with a C++ class. Profiling does need a considerable amount of memory for the profiling database which is kept in memory at all times. Therefore, if the physical memory is close to the amount of memory used by an un-profiled application, profiling will result in high swap activity; the execution speed of an application will deteriorate by a few orders of magnitude.

4. DESIGN

The design objective is to create a tool that will combine non-intrusive performance measurement for Session Beans and/or Entity Beans using interceptors with a JDBC driver wrapper. A JDBC Driver wrapper will be used to collect and analyze performance statistics and JDBC queries of EJB based systems using an open source application server, JBoss. The system will support continual monitoring of a deployed application and will help to identify performance bottlenecks as well as have ability to capture and display SQL statements, queries, and batch operations that are sent to any JDBC-compliant database driver along with performance analysis.

The first part of the performance measurement tool will extract performance details of the EJB system after it has been deployed. Such non-intrusive performance instrumentation can be useful in the tuning and porting stages of an EJB based project when there is a need to understand the impact of a new operating environment without having full access to the system's source code. The tool will make it unnecessary to insert performance measurement code into the application.

The second part of the tool will display in real time the queries going to the database. The tool will pass the calls issued by an application to an underlying JDBC driver and capture detailed information about the calls. All parameters and function results for JDBC calls can be captured and displayed. The monitoring can be enabled without changing the application. The tool helps to simplify performance and tracking of JDBC statements issued by an application. The JDBC tracking will not require any changes in the application source code and will be easily configured by changing a property file.

A Swing based GUI will be implemented to display various performance data such as average, minimum, maximum and count, as well as database queries with execution time. This GUI will display total time spent in selected Session and Entity Bean methods and provide a visual analysis of the selected execution layer where the majority of the time was spent.

The uniqueness of the system lies in its combination of JDBC query tracking with continual performance analysis capabilities. Once deployed and run, the system will be able to identify performance bottlenecks in Session or Entity Beans and point out the time spent in JDBC interaction and/or business logic.

5. SYSTEM ARCHITECTURE

The overall system architecture and major event-flow is shown in Figure 5. All the remote or local calls performed by the client are immediately trapped by a combination of Home Interceptor and Method Interceptor.

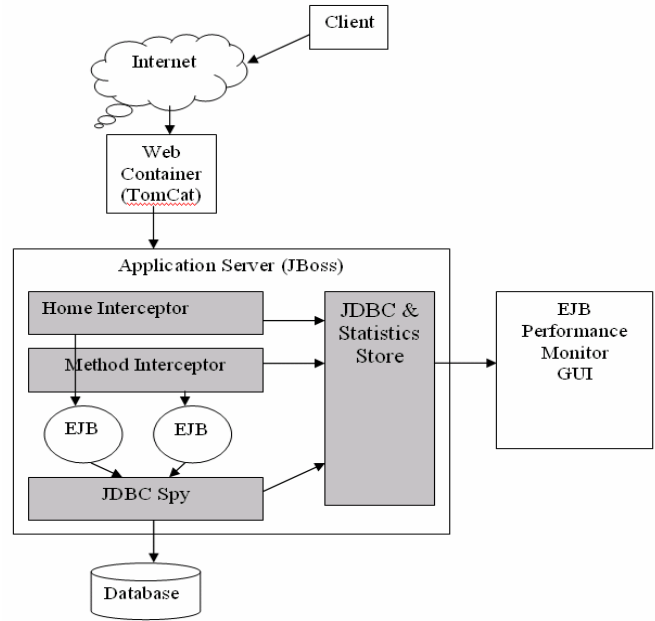


Figure 5: System Interaction

Flow of Events:

1. A Client (through a browser) comes to the Web Container.
2. Web Container calls an application server to retrieve a home reference and EJB object.
3. Home and EJB Object calls are intercepted by Home and Method Interceptors and performance information is put into Statistics Store.
4. Session or Entity Bean executes JDBC calls.
5. Each JDBC call is intercepted by JDBC Spy and performance information is logged into Statistics Store.
6. EJB Performance Monitor GUI retrieves the statistics data.

Sequence of steps:

1. Client obtains EJB Home Reference. Home Interceptor catches the call and measures the invocation time.
2. EJB Object is created by using Home reference.
3. Local or remote method is executed on EJB reference. Proxy Interceptor catches the call and measures invocation time.
4. Business method is executed.

A typical EJB application executes a number of queries and statements that are passed to a JDBC driver (which is typically provided by the vendor or an independent party) and then passed to the database. More often than not, inefficient, redundant or expensive database queries and statements become a major bottleneck. In order to simplify performance and tracking of JDBC statements, the JDBC catcher portion of the application is introduced.

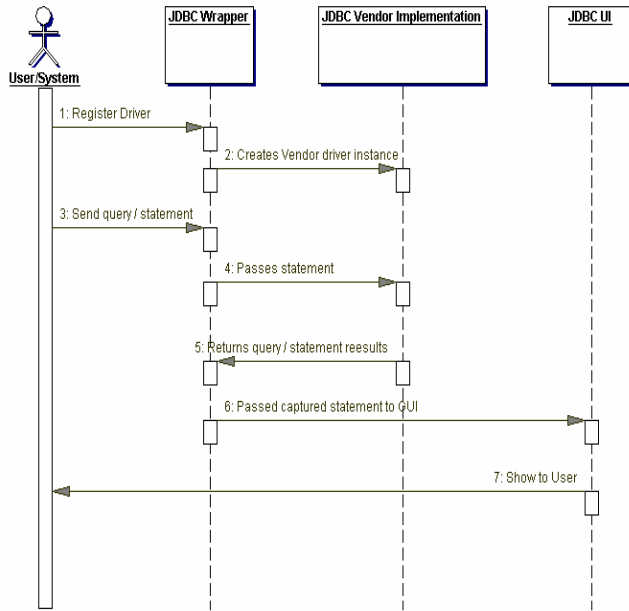


Figure 6: JDBC Wrapper Sequence Diagram

Sequence of steps:

1. User or system registers JDBC wrapper as a standard JDBC Driver.
2. JDBC Wrapper passes all instantiation parameters to specific vendor implementation and creates a “pass-through” driver object.
3. The user or the system issues a JDBC request (For example, Statement, PreparedStatement, Batch, etc).
4. JDBC Wrapper captures the statement and all of its bind variables and passes that statement to the “pass-through” driver instance, which actually executes it.
5. Vendor implementation returns result of the query / statement.
6. JDBC Wrapper passes captured statement and all of its information, such as bind variables, execution time, and number of rows in the result, to the GUI client through a network socket.

GUI Client displays all the captured information to the user.

6. RESULTS AND TESTS

The test case environment consisted of the following:

- A. Operating System: Windows 2000
- B. JDK 1.4
- C. Application Server: JBoss 3.2.6
- D. Database: Oracle 9.2.0.5

All the tests were run using a client to an EJB system executing three facade based Session Beans with remote methods 100 times. All methods used JDBC interaction.

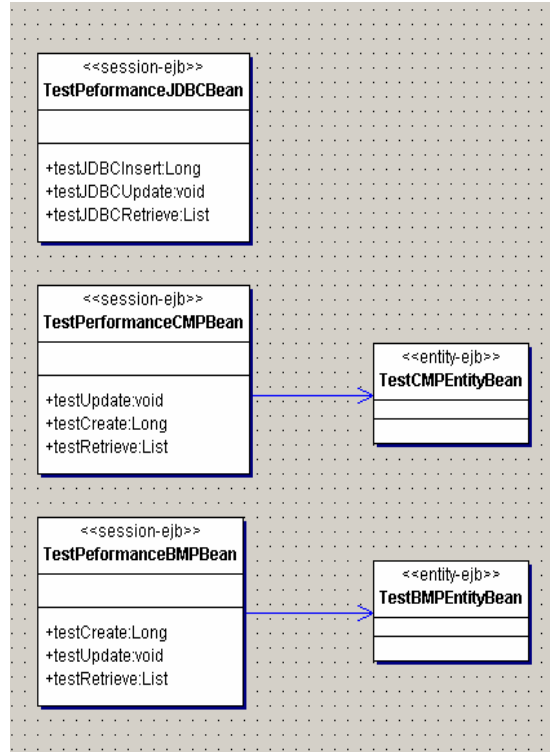


Figure 7: Test Session Beans

6.1 CPU TESTS

First, a set of test cases was executed to test execution time of the system as a whole. The following CPU based tests cases were run:

1. Interceptor plus JDBC Monitor. The system was run with the Interceptor collecting the statistics and the JDBC Spy driver. This setting has the highest level of overhead since the Interceptor collects method invocation statistics and the JDBC Spy intercepts JDBC calls.
2. A Clean System without any profiling tools performs in 15.66 (ms) versus 21.18 (ms) for a system that has both the Interceptor and the JDBC Spy driver.
3. Interceptor Only. This test measures the overhead of the Interceptor system that is designed to measure method invocation statistics. While a clean system performs the method invocations within 15.66 ms; the Interceptor based system does it by 17.8 ms on average.
4. JProfiler. The system was run with the JProfiler monitoring tool. The stress that is put on the system by JProfiler is clearly evident. The average response time went up to 46 ms versus 15 ms for the original system. The difference is almost a factor of 3.

CPU Test Summary:

	Interceptor & JDBC Spy(ms)	Interceptor Only (ms)	JProfiler (ms)	Clean System (ms)
AVG	21.18	17.8	45.46	15.66

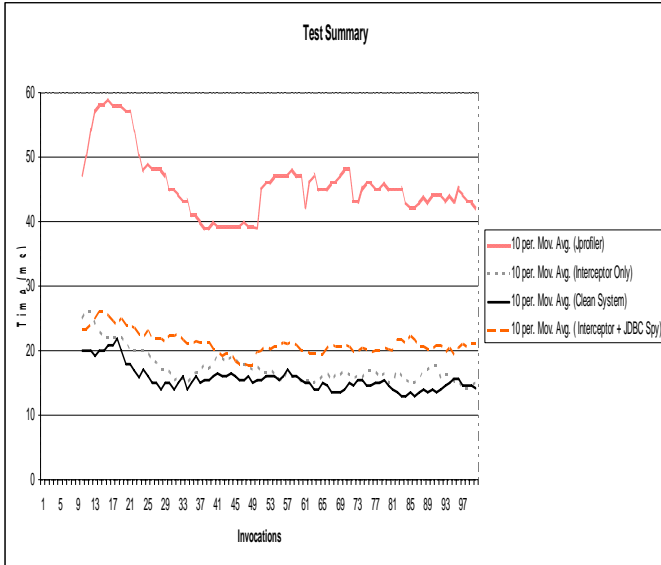


Figure 8: CPU Test Summary

6.2 Memory Tests

The first test was run on the system with both the Interceptor and the JDBC Spy enabled. The tests were run for 5 hours with a client request made every 5 minutes. The memory measurements were taken every 15 minutes.

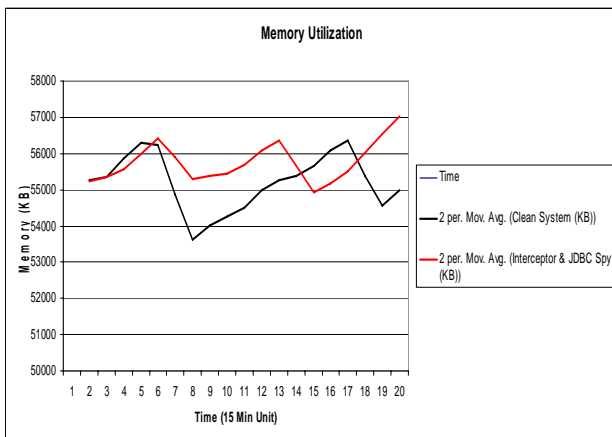


Figure 9: Memory Utilization of Interceptor and JDBC Spy vs. Clean System

JProfiler crashed the system repeatedly with OutOfMemory errors. This demonstrates the large amount of memory required by JProfiler.

6.3 GRAPHICAL USER INTERFACE

The Graphical User Interface features an “Outlook” like panel design where the screen selection is made on the left, and the corresponding screen is shown on the right on the main panel.

Once the connection is established, the following screen, as shown in Figure 10, is displayed to the user:

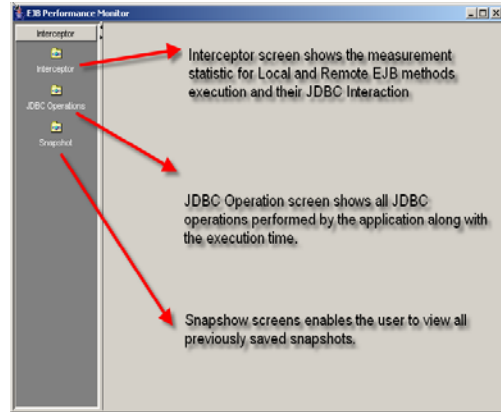


Figure 10: Main Application View

The screens are separated by functionality into three categories:

1. Interceptor Screen that shows statistics for Local and Remote EJB methods, and corresponding JDBC calls.
2. JDBC Operation Screen that shows all the JDBC calls performed by the application with execution time associated with each call.
3. Snapshot Screens that can be used to view previously saved performance statistics for the application.

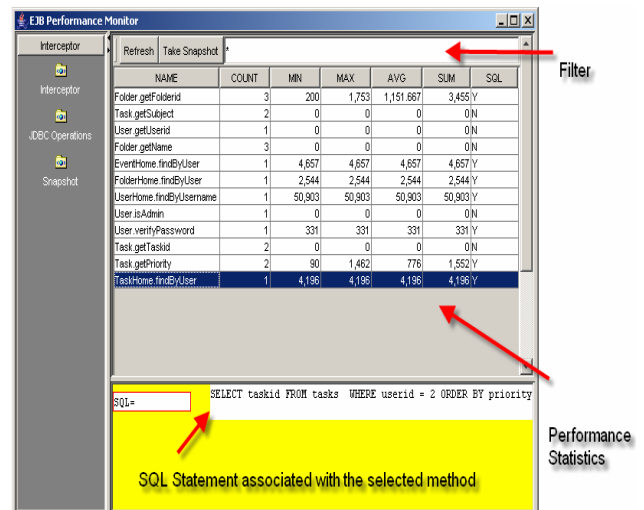


Figure 11: Interceptor Screen

The Interceptor Screen performs the following functions:

1. Displays performance statistics for all EJB Local and Remote methods that were intercepted on the server side.
2. Displays JDBC Statements/Queries that were executed by the selected method or indicates that the selected method did not perform any JDBC activity.
3. Allows filtering on the presented content to limit the data to a particular class or set of classes.
4. Takes a snapshot of current performance statistics and saves it to a comma separated value file. This file can be retrieved later and read by the Snapshot Screen.
5. Features a Refresh button that updates the current data set from the server.

SQL	TIME
SELECT userid FROM arusers WHERE username = 'denis'	2544
SELECT * FROM arusers WHERE userid = 2	231
SELECT folderid FROM folders WHERE userid = 2	771
SELECT taskid FROM tasks WHERE userid = 2 ORDER BY priority	1141
SELECT eventid FROM events WHERE userid = 2 AND mydate > '2005-05-01' ORDER ...	1562
SELECT * FROM tasks WHERE taskid = 1	70
SELECT * FROM tasks WHERE taskid = 1114333957243	120
SELECT * FROM folders WHERE folderid = 1111883894038	120
SELECT * FROM folders WHERE folderid = 1113183939317	30
SELECT * FROM folders WHERE folderid = 1113184090334	211

SQL: SELECT userid FROM arusers WHERE username = 'denis'

Time ms: 2544

Figure 12: JDBC Operation Screen

The JDBC Operation Screen performs the following functions:

1. Displays all executed JDBC Statements or Queries.
2. Shows the time spent in each JDBC call.

7. CONCLUSION AND FUTURE WORK

Where do performance issues come from? There are plenty of answers to that question: inefficient design, extensive serialization and de-serialization, a large number of database operations, or poor resource allocation and usage.

Performance measurement is often considered at the last moment because performance problems often do not occur until the system is completely developed and deployed. Finding a performance bottleneck at this point can be a very hard task to accomplish. The problem of detecting the source of the performance issue, already difficult in test systems, becomes even more challenging in the production environment. However, once the exact cause of a performance problem is known, it is often relatively easy to fix. The solution could involve rewriting the code to take advantage of an EJB container cache, changing the design to avoid extensive object creation, or clustering the system.

The proposed software infrastructure can be a great help for performance measurement in an EJB environment. The prototypical implementation was used to collect performance data in a simulated web site. The focus of the prototypical implementation was to demonstrate that such a low overhead, non-intrusive implementation can be built and used in a production environment where a high amount of overhead cannot be tolerated. The implementation also covered the needs of developers to obtain the information at the method object level of the system without being overwhelmed by needless details. Another important aspect of the prototypical implementation is its ability to monitor an application's interaction with relational databases through JDBC drivers. The combination of object and database monitoring might be a first step to provide a composite performance model.

The proposed system can be further improved by providing a persistence mechanism for the collected results, call-tree visualization and various alerts. These improvements can be considered a part of the next step in creating a system measuring framework that facilitates dynamic performance optimization and uses on-line performance monitoring to determine when performance expectations are not complied with and recommends alternative components to be used.

REFERENCES

- [1] D. Bales, "Java Programming with Oracle JDBC", O'Reilly; First Edition, December 2001
- [2] R. Brunner, "Tuning Java Database Performance: Understanding the Role of the Driver", DevEx conference, October 21, 2002
- [3] R. Mohat, "Performance Measurement and Analysis of Database Interface Technologies: JDBC, EJB (CMP2.0) and Oracle Toplink", Arizona State University East, Division Of Computing Studies, May 2004
- [4] J. Shirazi, "Java Performance Tuning", O'Reilly, Second Edition, January 2003
- [5] J. Goodson, "Performance Tips for the Data Tier (JDBC)", TheServerSide.com, Enterprise Java Community, April 2004
- [6] Ian Gorton, Anna Liu, "Evaluating the Performance of EJB Components", IEEE Internet Computing, June 2003
- [7] Jenny Yan Liu, "Performance and Scalability Measurement of COTS EJB Technology", University of Sydney, School of Information Technologies, October 2002
- [8] Baskar Sridharan, Balakrishnan Dasarathy, Aditya P. Marthur, "Building Non-Intrusive Performance Instrumentation Blocks for CORBA-based Distributed Systems", IEEE Internet Computing, March 2000
- [9] Iron Grid Computing, "Driver manual", IronGrid Inc, April 2003
- [10] S. Huber, "SQL Profiler: Definitive Guide", Jahia Software, November 2003
- [11] Data Direct Connect Inc, "DataDirect Connect for JDBC Guide", DataDirect Technologies Inc, May 2004
- [12] Swiss SQL, "API manual", SwissDriver Inc, June 2004
- [13] W. Louth, "JDBInsight 2.1 manual", JInspired Inc, April 2004
- [14] Adrian Mos, John Murphy, "New Methods for Performance Monitoring of J2EE Application Servers" Performance Engineering Laboratory, School of Electronic Engineering, Dublin City University, Ireland.
- [15] JProfiler, "Driver manual", EJ-Technologies GmbH, August 2004
- [16] B. Sridharan, S. Mundkur and A. P. Mathur, "Nonintrusive Testing, Monitoring and Control of Distributed CORBA Objects", TOOLS Europe 2000, St. Malo, France, June 2000
- [17] F. Lange, R. Kroeger, M. Gergeleit, "JEWEL: Design and Measurement of a distributed Measurement System", IEEE Transactions on Parallel and Distributed Systems, November 1992.
- [18] Ed Roman, "Mastering Enterprise Java Beans", Published by John Wiley & Sons, Inc., October 2002