

# RUBiS Revisited: Why J2EE Benchmarking is Hard

Jaime Spacco and William Pugh  
Dept. of Computer Science, University of Maryland  
College Park, MD, 20742 USA  
{jspacco,pugh}@cs.umd.edu

## ABSTRACT

We have replicated the experiments of Cecchet et al. detailed in "Performance and Scalability of EJB Applications" at OOPSLA '02. We report on our experiences configuring, deploying and tuning Enterprise software. We found a number of configuration problems that suggest that some of the conclusions of the original work were wrong.

## 1. INTRODUCTION

Measuring the performance of J2EE applications is notoriously difficult, and the results are easily influenced by external factors because an Enterprise application typically consists of many software components that may interact with each other and the underlying hardware in unpredictable ways. Writing, deploying and tuning J2EE applications requires a diverse skillset that includes knowledge of systems, programming patterns, networking, clustering, transactions and databases.

In [1], Cecchet et al. present the Rice University Bidding System (RUBiS), an online auction site loosely modeled on www.eBay.com [2]. The basic functionality of RUBiS is implemented using several EJB container configurations in order to measure their scalability. The original RUBiS work tests these configurations using a variety of open source software products including JBoss [3], JOnAS [4], Tomcat [5] and Apache [6].

We have chosen to focus our efforts on JBoss (specifically JBoss-3.2.3) because JBoss cleanly incorporates everything necessary for an Enterprise installation— a web server, a servlet-engine (Tomcat), and a flexible and highly configurable EJB container. We have further focused our analysis on four implementations of RUBiS: SessionBeans only, Container-Managed Persistence (CMP 2.0) with local interfaces, Apache with PHP, and Servlets only. The other configurations, such as Servlets that manipulate Entity Beans directly, are not commonly used design patterns, and are therefore unlikely to appear in practice. Finally, we use a Sunfire 6800 with 24 processors and 72 GB of RAM for

the Application Server to perform EJB experiments and Apache's httpd-2.0.52 with PHP-4.3.9 to perform PHP experiments.

In order to achieve good performance using JBoss, we had to perform extensive tuning not only to JBoss but also to other software used by RUBiS. In the process, we discovered a number of problems with the original RUBiS configuration that negatively impacted performance. We detail these issues in this paper. In addition, we found a severe configuration problem with the Container-Managed Persistence settings used by RUBiS that we believe casts doubt on some of the original conclusions about the poor scalability of CMP.

## 2. CLIENT-SIDE PROFILING AND RESULTS GATHERING

The RUBiS software produces an abundance of extremely useful charts and graphs detailing processor, memory and network usage for the database host, middleware servers, and client machines. However, this information is collected using a series of ad-hoc scripts and unix utilities that proved to be difficult to use and maintain, and would likely be impossible to adapt to benchmark another framework.

We found that the difficulties caused by the ad-hoc nature of client-side software and results-gathering mechanisms lend tremendous support for a framework, such as CLIF [7], to simplify and standardize client-side benchmarking.

## 3. DATABASE

The overall data reported did not include extensive profiling of the traffic between the appserver and database, or of the internal state of the database. Using 'mysqladmin', we queried the database for its internal status at the end of each run. Using the information provided by this lightweight profiling mechanism, we discovered several unnecessary performance bottlenecks.

We first discovered several issues with the indexing of the database used by RUBiS. A relational database should typically maintain an index on columns that will be part of a 'WHERE' clause to speed up searches. There were four unindexed columns in the original RUBiS database that were frequently used by 'WHERE' clauses as part of search criteria. This negatively impacted performance for those queries.

We also discovered an important difference between the Java and PHP versions of the code (The PHP version of the code was not part of the original RUBiS experiment but rather was contributed by a RUBiS user). The PHP version

of RUBiS returned unsorted results for certain queries that were sorted in all the Java versions of the code. Initially, this appeared as if the PHP code was performing faster when upon closer inspection it became clear that PHP was merely requesting the database to perform less work. This is a very subtle problem because the PHP code and the Java code returned exactly the same data without signaling any errors; they merely differed on the *order* in which the data was returned.

## 4. GARBAGE COLLECTION

For long-running server applications that must respond to client requests with low latency, it is important to choose a garbage collection algorithm that minimizes pause times. In particular, full “stop-the-world” collections must be avoided at all costs. Luckily, modern JVMs such as Sun’s HotSpot JVM come equipped with a variety of garbage collection algorithms which can be carefully tuned to minimize pause times.

Unfortunately, no matter which algorithm we chose, we suffered a performance hit due to a mysterious “stop the world” garbage collection that occurred every 60 seconds.

It turns out that, by default, RMI applications running on Hotspot perform a full “stop the world” GC every 60 seconds in case weakly referenced remote objects have become garbage, *even if the VM has been specifically configured to minimize full GCs*. It is crucial to disable this feature in order to fully benefit from garbage collection algorithms designed to minimize pause times.

## 5. BUGS IN COMPONENT SOFTWARE

While finding bugs is not typically part of performance tuning, bugs often manifest themselves as performance quirks or irregularities. For example, we found a race condition in JBoss-3.2.3 where unsynchronized access to a WeakHashMap caused threads to get stuck in an infinite loop (This bug has since been fixed in JBoss-3.2.4).

We have also encountered problems where access to the MySQL database can cause corruption in the table structure that requires repair operations before the database can be accessed again.

This bug is infrequent but insidious because its occurrence can invalidate an entire batch of results. Upgrading MySQL to 4.1.5-gamma appears to decrease the frequency with which this bug manifests itself, but does not eliminate it completely.

In addition to problems with MySQL, we found the support for threads of the 2.4.20-8smp linux kernel to be inadequate for testing Enterprise software. The Linux JDKs of both Sun and IBM consistently crashed from internal signals raised by native thread libraries leading us to conclude that the problem was with linux rather than Java.

## 6. TRANSACTIONS, CACHING AND COMMIT OPTIONS

The EJB specification [8] details three commit options: A, B and C. Option A assumes that the appserver has exclusive write access to the underlying datastore, and thus can cache entity data. Commit Option B (the default for JBoss) uses a separate bean instance per transaction, does not perform caching, but does not require that the appserver

have exclusive write access to the datastore. Commit Option C is similar to B except that it stores unused beans in the passivated state rather than the ready state, and thus uses less memory than the other options. In addition, JBoss adds a Commit Option D, which is identical to Commit Option A except that cached data becomes stale and must be periodically refreshed.

In [9], Brebner and Ran provide an extensive analysis of the commit options and empirical results for different caching policies.

Using the light-weight profiling of the database we added to the RUBiS test framework, we noticed that the original CMP version of RUBiS generated as much as an order of magnitude more traffic between the appserver and the database than any of the other versions. We experimented with Commit Option A, and this seemed to fix the problem. However, caching entity data alone could not explain the difference in data sent.

The original CMP version of the code contained few transactions in the Session Facade layer but required transactions for all entity bean methods. Intuitively, this design maximizes concurrency (at the expense of correctness) and prevents the database from becoming the bottleneck of the experiment. Furthermore, the MyISAM table format used by RUBiS does not even provide full transactional support, so this configuration should not adversely affect performance.

However, under commit option B, this approach can have the unintended side-effect of generating up to an order of magnitude more traffic between the appserver and the database. This happens because, in the absence of transactions, the appserver performs a fresh SELECT for every value fetched from the database.

For example, suppose an Entity Bean E has two fields, a and b, which are represented by columns A and B in the database. With no transactions in the session beans but transactions in all of the entity bean methods, every time an E is loaded from a row of the database, the appserver performs a SELECT operation to fetch the value of column A, but pessimistically performs a second SELECT operation to fetch the value of column B because, from the appserver’s perspective, the state of the database could have changed after loading A but before loading B. This happens *even if the appserver already optimistically loaded the value for column B while fetching column A with the previous SELECT*.

This is in essence a race condition, though instead of causing an error, it causes an artificial performance bottleneck in terms of extra database traffic. This problem can be fixed by adding transactions to the session bean methods, or by using commit option A. Simply changing the commit option to A decreased the traffic between the appserver and the database by an order of magnitude and increased the throughput of CMP by a factor of 3.

## 7. RESULTS

In Figure 1 we present our results after tuning the configuration of RUBiS given all of the parameters discussed above.

Simply changing the commit mode from B to A (or alternately, adding transactions to the Session Facade) dramatically improves the performance of the CMP implementation. We show the CMP results using the original RUBiS settings (CMP-orig) in addition to CMP results after changing the commit option from B to A. Each implementation should

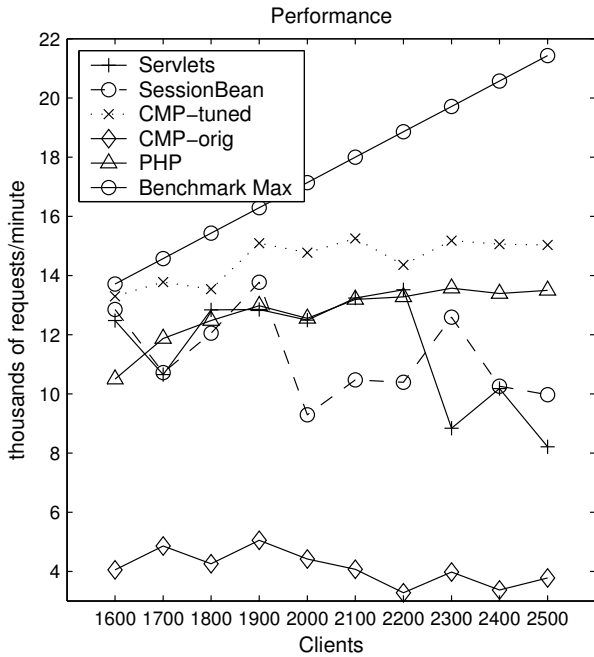


Figure 1:

benefit about equally from the other changes.

The y-axis of this chart represents thousands of handled requests per minute while the x-axis represents the number of clients.

We observe that with proper tuning, CMP2.0 can provide performance competitive with other implementations. This suggests that the conclusions of the original RUBiS work—that CMP offer substantially worse performance than other implementation strategies—are not conclusive.

RUBiS only provided the min, max and average times for a client's transitions between different web pages, or states. In addition, the number reported for each result was throughput measured in requests processed per minute. While raw throughput is perhaps the most important measurement of performance for an Enterprise system, it is not the only measure of performance. For example, in [10], Welsh et al discuss different quality-of-service measurements.

We stored the entire set of transition times in order to make some simple quality-of-service measurements. We present the results for PHP in 2, servlets in 3, session beans in 4, and CMP2.0 in 5. We show the maximum, mean and median times, as well as the percentage of transitions that required more than 1 second. What we found is that even where the raw throughput is virtually identical, the breakdown of maximum, mean, median and percentage of outliers can vary widely between implementations. For example, the servlets-only implementation has very few transitions that require over 1 second while the other implementations have as many as 40% of their transitions take over 1 second. This suggests that the servlets only implementation will scale better when we ultimately run experiments with larger numbers of clients.

These distinctions can be very important depending on the requirements and typical workload of an Enterprise system.

| clients | max    | mean | median | over 1 sec |
|---------|--------|------|--------|------------|
| 1100    | 3.18   | 0.05 | 0.02   | 0.4%       |
| 1200    | 4.62   | 0.06 | 0.02   | 1.2%       |
| 1300    | 3.07   | 0.04 | 0.02   | 0.1%       |
| 1400    | 4.25   | 0.06 | 0.02   | 0.7%       |
| 1500    | 21.11  | 0.09 | 0.02   | 0.9%       |
| 1600    | 24.11  | 0.16 | 0.04   | 2.1%       |
| 1700    | 46.43  | 0.62 | 0.23   | 13.0%      |
| 1800    | 195.71 | 1.43 | 0.44   | 36.6%      |
| 1900    | 393.99 | 2.17 | 0.53   | 41.1%      |
| 2000    | 98.5   | 1.37 | 0.6    | 34.0%      |

Figure 2: PHP quality-of-service measurements

| clients | max    | mean | median | over 1 sec |
|---------|--------|------|--------|------------|
| 1100    | 8.4    | 0.04 | 0.01   | 0.5%       |
| 1200    | 991.94 | 0.93 | 0.02   | 11.6%      |
| 1300    | 882.75 | 0.18 | 0.01   | 0.2%       |
| 1400    | 269.5  | 0.13 | 0.01   | 0.9%       |
| 1500    |        |      |        |            |
| 1600    | 259.79 | 0.3  | 0.02   | 1.4%       |
| 1700    |        |      |        |            |
| 1800    | 278.79 | 0.79 | 0.03   | 3.4%       |
| 1900    | 400.51 | 0.84 | 0.05   | 4.2%       |
| 2000    | 736.17 | 0.59 | 0.01   | 2.2%       |

Figure 3: Servlets quality-of-service measurements

| clients | max    | mean | median | over 1 sec |
|---------|--------|------|--------|------------|
| 1100    | 5.29   | 0.07 | 0.01   | 1.1%       |
| 1200    | 1.44   | 0.02 | 0.01   | 0.0%       |
| 1300    | 12.12  | 0.04 | 0.01   | 0.8%       |
| 1400    | 6.8    | 0.08 | 0.02   | 0.8%       |
| 1500    | 3.07   | 0.03 | 0.01   | 0.4%       |
| 1600    | 190.31 | 0.59 | 0.05   | 8.3%       |
| 1700    | 7.95   | 0.14 | 0.04   | 1.7%       |
| 1800    | 29.47  | 0.37 | 0.08   | 6.6%       |
| 1900    |        |      |        |            |
| 2000    | 46.41  | 0.81 | 0.32   | 21.6%      |

Figure 4: Session Bean quality-of-service measurements

| clients | max   | mean | median | over 1 sec |
|---------|-------|------|--------|------------|
| 1100    | 3.1   | 0.02 | 0.01   | 0.1%       |
| 1200    | 10.69 | 0.03 | 0.01   | 0.3%       |
| 1300    | 11.39 | 0.03 | 0.01   | 0.4%       |
| 1400    | 10.16 | 0.04 | 0.01   | 0.7%       |
| 1500    | 1.3   | 0.02 | 0.01   | 0.0%       |
| 1600    | 13.88 | 0.17 | 0.02   | 4.8%       |
| 1700    | 45.07 | 0.51 | 0.02   | 10.1%      |
| 1800    | 32.12 | 0.67 | 0.03   | 15.6%      |
| 1900    | 50.67 | 1.19 | 0.03   | 20.5%      |
| 2000    | 50.6  | 1.57 | 0.03   | 22.7%      |

Figure 5: EJB CMP quality-of-service measurements

## 8. CONCLUSIONS

Our results suggest that some of the conclusions reached by Cecchet et al are not true in general. For example, the original RUBiS work illustrated that Servlets with SQL scales better than SessionBeans with SQL, which in turn scales far better than any of the designs using CMP. We feel that the lack of tuning for the appservers coupled with the relative immaturity of JBoss and the CMP 1.1 standard suggests that their conclusions are no longer valid. In our work, we have observed that after proper tuning, the PHP, SessionBeans, Servlets and CMP implementations of RUBiS scale similarly.

More importantly, our work illustrates that performance numbers for J2EE applications can be dramatically influenced by tuning and tweaking the configuration parameters to several different software components on the J2EE stack. It is also difficult to predict the effects of composing the various tweaks and optimizations.

Further generalizing these results is extremely problematic. Benchmarks are simple, artificial pieces of a “real world” problem that by their nature can be difficult to generalize.

However, this is especially true in the case of Enterprise computing, where it is generally not possible to scale down the “real world” conditions to work in the laboratory. At JavaOne 2003 [11], the lead architects for eBay 3.0 detail the scale of their application— 69 million users, 34 million active users in Q1 of 2003, 16 million items spread across 28,000 categories, 30,000 lines of code changes per week, and 1 billion dynamic page requests per day split between 1200 URLs.

While not all enterprise applications have such rigorous requirements, it is nonetheless important to consider the extreme demands placed on these systems and the radical differences between a production system and a system being benchmarked in a laboratory setting.

## 9. FUTURE WORK

We want to continue our experiments for larger numbers of clients because we have not yet managed to stress the middleware enough to lower the throughput substantially below the theoretical maximum.

We would like to test RUBiS using JOnAS, a different open source application server that statically pre-compile the EJB classes. We would also like to test Apache’s standalone version of Tomcat in place of JBoss’s embedded Tomcat. There are also many other tunable features of JBoss, such as PreparedStatement caching, bean and thread pool sizes, and pre-fetching of entity data, that we should look into.

Another logical step is examining other Enterprise benchmarks such as SpecJAppserver 2004.

## 10. REFERENCES

- [1] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 246–261, 2002.
- [2] eBay. <http://www.ebay.com>.
- [3] JBoss Application Server. <http://www.jboss.org>.
- [4] Java Open Application Server. <http://jonas.objectweb.org>.
- [5] Jakarta Tomcat. <http://jakarta.apache.org>.
- [6] The Apache HTTPD Server Project. <http://httpd.apache.org>.
- [7] Emmanuel Cecchet and Bruno Dillenseger. Clif is a load injection framework. In *Middleware Benchmarking: Approaches, Results, Experiences*, 2003.
- [8] EJB 2.1 specification. <http://java.sun.com/products/ejb/index.jsp>.
- [9] Paul Brebner and Shuping Ran. Entity bean a,b,c’s: Enterprise java beans commit options and caching. In R. Guerraoui, editor, *Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 36–56. Springer, 2001.
- [10] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243. ACM Press, 2001.
- [11] One Billion Transactions a Day: A Java[tm] 2 Platform, Enterprise Edition (J2EE[tm]) System Architected with the Core J2EE Patterns. <http://javaonline.mentorware.net/>.