# Performance Assessment of Web Services in the STEP Framework

Miguel L. Pardal, Joana P. Pardal, and José Alves Marques

Department of Computer Science and Engineering
Instituto Superior Técnico, Technical University of Lisbon
Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
{miguel.pardal,joana.paulo.pardal}@ist.utl.pt,
jose.marques@link.pt

**Abstract.** This chapter presents a performance study of the STEP Framework, an open-source application framework implemented on the Java platform that uses many popular open-source libraries, including: Hibernate, JAX-WS, and Log4J. This framework has been used for several years to teach development of distributed enterprise applications to undergrad students. This chapter also describes the performance measurements over a flight reservation web service that is included as an example in the source code distribution. It presents an assessment of the web service and shows how the performance of this specific application was studied in detail. The achieved results are put in context and compared with other technologies, highlighting the existing trade-offs.

**Keywords:** Web Services, Performance, Measurement.

## 1 Introduction

Enterprise applications have many demanding requirements [1], and some of the most important are related to *performance*. Performance analysis is a challenge [2] [3], that can be especially hard for inexperienced developers. To verify if an implementation is performing as expected, run-time data must be collected and analyzed. This data can be used to compare design and configuration alternatives. However, collecting such data in the application requires many modifications to the original source code.

The *Simple, Extensible, and for Teaching Purposes (STEP) Framework*[1] [4] is an open-source application framework. Its source code is intended to be small and simple enough to allow any developer to read and understand it thoroughly. The goal is to learn how the architectural layers are implemented in practice and to be able to change small details that are usually hidden in professional frameworks. This is especially important for students. The collected metrics allow them to better understand the existing trade-offs of alternative approaches. In fact, this framework has been used for several years in 'Software Engineering' and 'Distributed Systems' courses lectured at Instituto Superior Técnico (IST), Technical University of Lisbon, to teach Computer Science and Engineering undergrad students how to develop Web Services with enterprise-like requirements.

[1] http://stepframework.sourceforge.net/

I. Ivanov et al. (Eds.): CLOSER 2012, CCIS 367, pp. 200–214, 2013.

Before the improvements we describe here, the STEP framework did not provide means to collect run-time data for later analysis. With this work the framework was extended with monitoring and analysis tools that enable developers to collect actual performance data and to use it to study how different decisions impact the overall performance.

What follows is a brief overview of the STEP framework architecture, followed by the description of a *performance assessment study*, detailing the new added tools and the results of the conducted experiments over a flight reservation web service that is available in the distribution.

## 2 STEP Framework Overview

The STEP Framework is a multi-layer, Java-based, enterprise-like application framework. It can be used to develop Servlet/JSP Web Applications and Web Services.

### 2.1 Architecture

The STEP Framework defines a typical layered architecture [1]. The main layers are *Domain* and *Service*. There are also *Persistence*, *View*, *Presentation* and *Web Service* layers. Each layer considers different implementation concerns.

The *Domain* layer is where an object-oriented solution for the requirements is implemented. Domain objects are persisted to a database using object-relational mapping through the Hibernate[2] library and its annotations.

The *Service* layer provides access to the application's functionalities through service objects, that access the domain objects, isolating them from upper layers, and managing transactions to ensure atomic, consistent, isolated, and durable (ACID) persistence.

The *View* layer provides Data Transfer Objects (DTO) that are used as input and output for service objects and uses JAX-B[3] technology.

The *Presentation* layer is responsible for user interaction through a Web interface, implemented with servlets and Java Server Pages (JSP). It uses Stripes[4] but there are also STEP variants using Struts[5] and the Google Web Toolkit[6].

There is a *Web Services* (WS) layer that provides remote access to services, using JAX-WS[7] technology.

STEP supports *Extensions* [5][4], a mechanism for intercepting the Service and Web Service layers that simplifies the implementation of cross-cutting concerns. Extensions proved very useful for implementing the performance monitors described later in the chapter.

A STEP development branch, called SmartSTEP [6] supports WS-Policy-like automatic configuration of Extensions to provide security, reliable messaging, logging, etc; as required by parties communicating with WS.

[2] `http://www.hibernate.org/`
[3] `https://jaxb.dev.java.net/`
[4] `http://www.stripesframework.org/`
[5] `http://struts.apache.org/`
[6] `https://developers.google.com/web-toolkit/`
[7] `https://jax-ws.dev.java.net/`

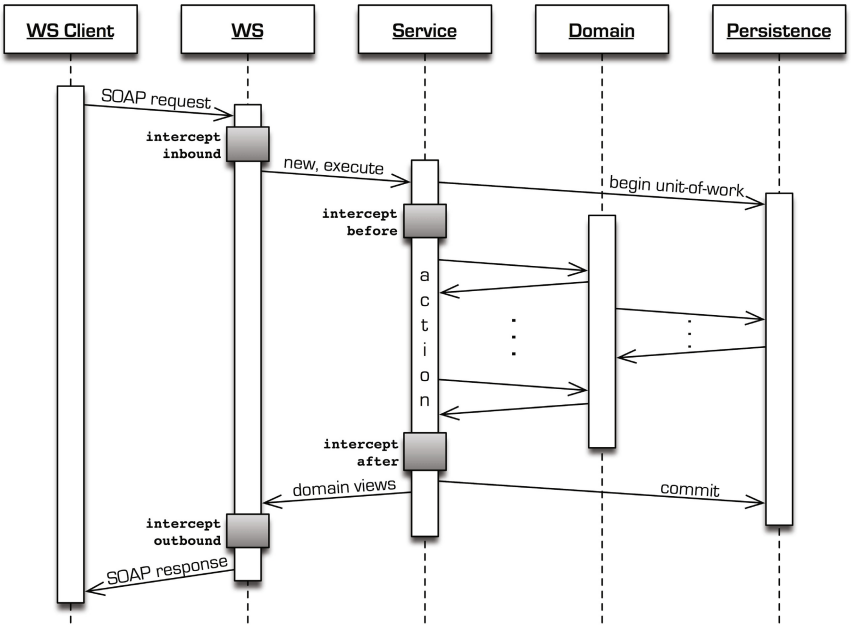


**Fig. 1.** Sequence diagram of a STEP Web Service invocation

### 2.2 Request Processing

The processing sequence of a request for a STEP Web Service is shown in Figure 1. A request begins in the client application (*WS Client*) that sends a SOAP envelope in an HTTP request to the server (*WS*). The application container at the server assigns a thread to execute the request from start to finish. The HTTP request is interpreted and dispatched to an instance of the JAX-WS servlet. The *WS* layer parses the SOAP envelope. The payload is deserialized from XML to Java objects using JAX-B.

The *Service* layer receives the view objects, starts an implicit database transaction, and invokes one or more domain objects. The *Domain* layer implements business logic using entity and relationship objects. The *Persistence* layer maps entities and relationships to database tables and vice-versa. SQL queries are generated and executed automatically by Hibernate. When the application-specific logic is complete, and if no error is reported, the *Service* layer commits the database transaction. Otherwise, the transaction is aborted and an error is returned. The resulting views (either the required results or the error message) are created and returned to the *WS* layer. The response payload is serialized from Java objects to XML. The JAX-WS servlet sends the SOAP envelope back to the client in the HTTP response. The request thread is typically returned to a thread pool, for later reuse. Several requests can be executed in parallel.

## 3 Performance Tools

The goal of the performance assessment tools is to breakdown the overall processing time, to identify the parts of the application that are worth improving.

Performance measurement tools can be classified as tracers and profilers [2]. A tracer [7] is a component that intercepts application code to record typed time-stamped events. Examples of tracing tools include libraries like Perf4J[8]. A profiler [8] is a program that monitors an application to determine the frequency of execution in specific code regions. A profiler can operate using sampling (application is interrupted periodically and measurements are taken), hardware counters (processor stores application performance data), or instrumentation (application source or binary code is augmented). Overall, sampling is faster but less accurate. There are several profiling tools available that combine the approaches mentioned above, like JProfiler[9] and YourKit[10]. Tracer are more lightweight than profilers because the latter require more complex interactions with applications [9]. Also, profilers are usually harder to use for server-side applications that have to handle multiple concurrent requests.

### 3.1 Our Approach

The performance tools for the STEP Framework follow the tracer approach. The goal was to collect run-time data, to analyze it, and to test performance improvement hypotheses. The main metric used was *request processing time* to measure (and improve) responsiveness.

The performance of Java programs is affected by application, inputs, virtual machine, garbage collector, heap size, and underlying operating system. All these factors produce random errors in measurements that are unpredictable, non-deterministic, and unbiased [10]. To quantify the random errors in measurements, the program runs had to be repeated several times. The presented values are the *mean* of the samples with a confidence interval (margin of error) computed with a confidence level of 90%, 95%, or 99%. At least 30 runs were executed for each program variation, so that the calculation of the confidence level could assume a normal distribution of the samples, according to the Central Limit Theorem [11]. Only changes in values greater than the error margin were considered statistically relevant and not the effect of random errors.

The performance analysis process encompasses all activities necessary to generate, collect, and analyze performance-related data. Figure 2 presents the data-flow diagram of our approach. Each activity is performed by a specific tool: *Domain Data Generator*, *Load Generator*, *Load Executor*, *Monitor*, *Analyzer*, and *Report Generator*.

The *Domain Data Generator* tool populates the database with realistic data, both in values and in size. The data population was realized using Groovy[11] scripts that parsed data files with domain descriptions and accessed the database to insert them.

The *Load Generator* tool produces files with serialized request objects, following templates for normal and error situations, creating loads that can be reproduced later.

The *Load Executor* tool was programmed to send requests. The script opens an object stream, reads request objects from it, and executes the operations: think (wait), search

[8] `http://perf4j.codehaus.org/`
[9] `http://www.ej-technologies.com/products/jprofiler/overview.html`
[10] `http://www.yourkit.com/`
[11] `http://groovy.codehaus.org/`

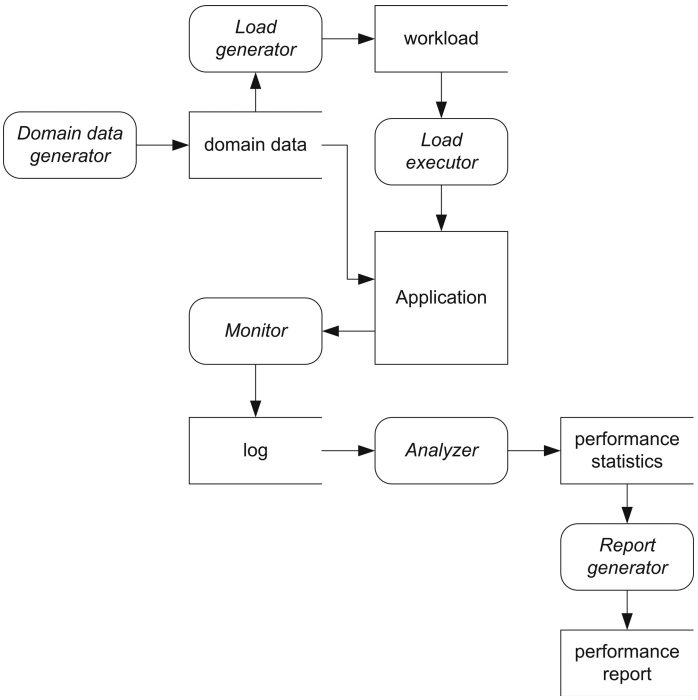


**Fig. 2.** Performance tool chain data flow diagram

flights, create single reservation, and create multiple reservations. The requests are sent to the specified WS endpoint. If an error is caught, the output message is logged, and the processing continues. This tool uses a thread pool of fixed size implemented with the java.util.concurrent package to run simultaneous virtual users and there is one thread for each simulated user.

The *Monitor* is the core component of performance analysis. When enabled, it collects request processing times for each architectural layer. It intercepts request processing at relevant interception points (represented in Figure 1, using grey boxes at the WS, Service and Persistence layers). Each specific interception point inserts measurement code. STEP extensions are used to intercept both the Service and the WS layers.

The *Analyzer* takes all samples of execution data resulting from multiple runs using the same settings, and computes sample statistics. A complete records file is summarized in a single row. For each numeric field, the mean, standard deviation, upper quartile, median, and lower quartile are computed. Finally, the overall statistics are computed. A similar procedure is applied to the virtual user output logs to produce error statistics from the WS client perspective that is the most relevant one for quality of service purposes.

Finally, the *Report Generator* uses the statistical data produced by the Analyzer and uses it to produce custom reports. For a more in-depth description of these tools, see our paper on the topic[12].

# 4 Experiments

Several experiments were conducted using the performance analysis tool chain, to identify performance problems and to propose solutions. The results are presented and discussed in this section.

## 4.1 Scenario System

The analyzed system was the *"Flight reservation Web Service"* (Flight WS) that is one of the example applications included in the STEP Framework source code distribution.

The initial Flight WS had only one operation: "create low price reservation". The following additional operations were developed: "search flights", "create single reservation", and "create multiple reservations". The reason for adding new operations was to allow more diverse kinds of requests using the most common data types (text, numeric, date, currency, and collections) and with different message sizes.

With the new operations it became possible to instantiate all the message archetypes defined in the JWSPerf Web Service benchmark [13], making Flight WS a typical Web Service. To a limited degree, conclusions made using Flight WS can be extrapolated to other WS with similar software architecture and user loads.

## 4.2 Hardware and Software Platform

The following machines and networks were used for the test runs.

**Machine A** with a Quad-core[12] CPU running at 2.50 GHz, 3.25 GB of usable RAM, and 1 TiB hard disk. It ran 32-bit Windows 7 (version 6.1.7600), MySQL 5.1.43, Java Developer Kit 1.6.0_18, Groovy 1.7.3, Apache Tomcat 6.0.14 and STEP 1.3.3 (includes Hibernate 3.3.2.GA, JAX-B 2.1.10, JAX-WS 2.1.7, Stripes 1.5.1).

**Machine B** with a Dual-core[13] CPU running at 2.53 GHz, 3 GB of RAM, and 500 GiB of hard disk storage. It ran the same software.

The machines were connected either by a 100 Mbit LAN or by a 10 Mbit LAN. The machines were configured to disable all system maintenance activities. The measurements were taken for the application's steady-state performance and not for start-up performance, since we are concerned with the running application's response times. Garbage collection and object finalization were considered as part of the steady-state server workload [14]. Unless otherwise stated, all the presented results were produced running in Machine A.

## 4.3 Request Time Breakdown

Table 1 presents the request processing time breakdown. Figure 3 represents the same data graphically.

[12] Intel Core 2 Quad CPU Q8300.
[13] Intel Core 2 Duo CPU P9500.

**Table 1.** Request processing time breakdown

| Slice | Time (ms) | Time % |
|---|---|---|
| Web | 2.83 | 0.98 |
| Web Service | 14.33 | 4.94 |
| Service | 203.14 | 70.07 |
| Hibernate Engine | 40.97 | 14.13 |
| Hibernate Writes | 15.52 | 5.35 |
| Hibernate Reads | 13.10 | 4.52 |

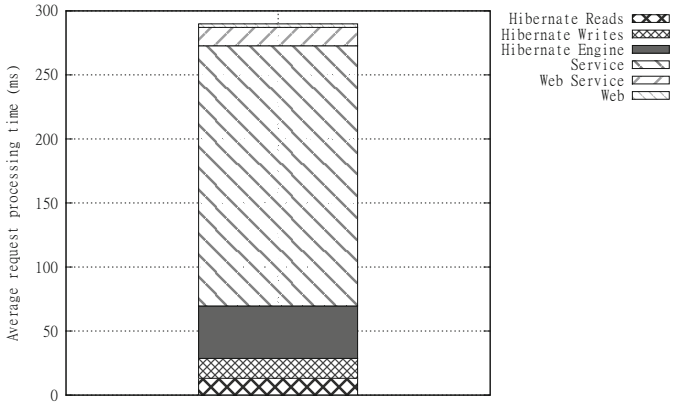


**Fig. 3.** Request processing time breakdown

The largest time slice is *Service* (70%) because it includes all the application-specific logic and also because it is the slice where the remaining – not specific to any layer – processing time is accounted for. The second largest slice is *Hibernates* (24%) as it manages the domain objects in the database. The *Hibernate engine* slice is significant (14%) because it includes when data is actually written to the database, at transaction commit time. The absolute value of roughly 300 milliseconds average processing time is only useful to compare with other measurements made in the same machine.

### 4.4 Monitor Implementation Comparison

The STEP framework performance monitor [12] had several iterations. Each was an attempt to more accurately capture the performance data.

Table 2 and Figure 4 present a comparison of the results of the same workload executed but using different monitor implementations to capture data:

- Perf4J monitor raw records (Perf4J raw);
- Perf4J monitor with aggregated records (Perf4J agg);
- Event monitor (Event);
- Layer monitor without Hibernate wrapping (Layer -Hwrap);
- Layer monitor with Hibernate wrapping (Layer).

**Table 2.** Request processing time percentages of different performance monitors. Each row sums to 100% of time spent.

| Monitor | Web | WS | Svc | Hib Eng | Hib W | Hib R |
|---|---|---|---|---|---|---|
| Perf4J raw | 0.71 | 4.26 | 87.00 | 0.00 | 4.58 | 3.45 |
| Perf4J agg | 0.79 | 4.51 | 4.59 | 0.00 | 10.86 | 79.25 |
| Event | 0.99 | 5.15 | 83.74 | 0.00 | 5.39 | 4.72 |
| Layer -Hwrap | 0.89 | 4.82 | 85.17 | 0.00 | 4.82 | 4.30 |
| Layer | 0.98 | 4.94 | 70.07 | 14.13 | 5.35 | 4.52 |

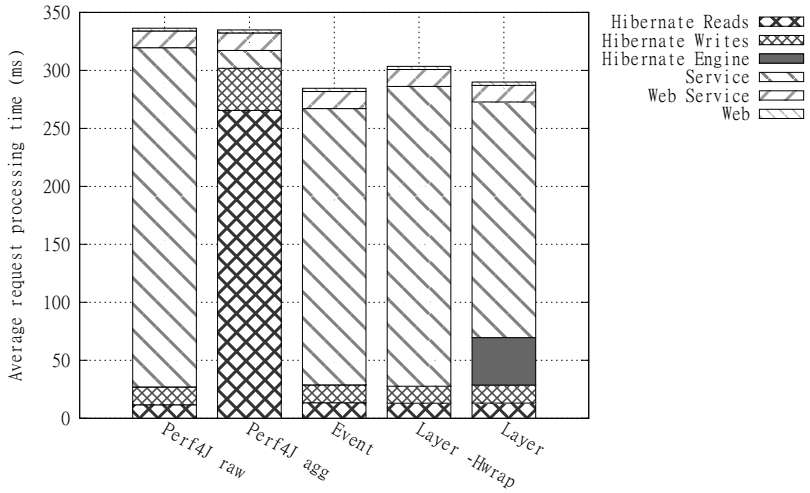


**Fig. 4.** Request processing breakdown of different performance monitors

The first choice for monitor was the Perf4J[14] library that uses stop-watch objects to time the execution of code blocks: on entry, the stop-watch is started; on exit, the stop-watch is stopped, timing of execution inside each layer. Perf4J delegates actual logging on the Apache Log4J library, already used by the STEP Framework. The performance events are logged in a separate log file and each stop-watch record has a start, time, tag, and (optional) message.

At first glance Perf4J was assumed to be underestimating the value of the Hibernate slice. Especially because the performance log files had (literally) thousands of lines stating that the time spent to load an object was 0 ms. These values were due to excessively fine-grained measurement of Hibernate calls. In practice, each call was too short to be accurately measured.

In the Perf4J monitor with aggregated records (*Perf4J agg*) the consecutive 0 ms records were combined and the elapsed time was computed using the time-stamps. The result of this mitigation attempt was a gross overestimation of the Hibernate slice, as confirmed by the other monitors. The mitigation failure was confirmed also by many

[14] `http://perf4j.codehaus.org/`

**Table 3.** Request processing breakdown for different request types, in percentages

| Request | Web | WS | Service | Hib Eng | Hib W | Hib R |
|---|---|---|---|---|---|---|
| All | 0.98 | 4.94 | 70.07 | 14.13 | 5.35 | 4.52 |
| Searches | 1.25 | 8.75 | 74.15 | 11.31 | 0.00 | 4.55 |
| Reservations | 0.71 | 0.83 | 62.69 | 19.37 | 12.17 | 4.23 |
| Faults | 0.83 | 4.60 | 86.73 | 1.96 | 0.00 | 5.89 |

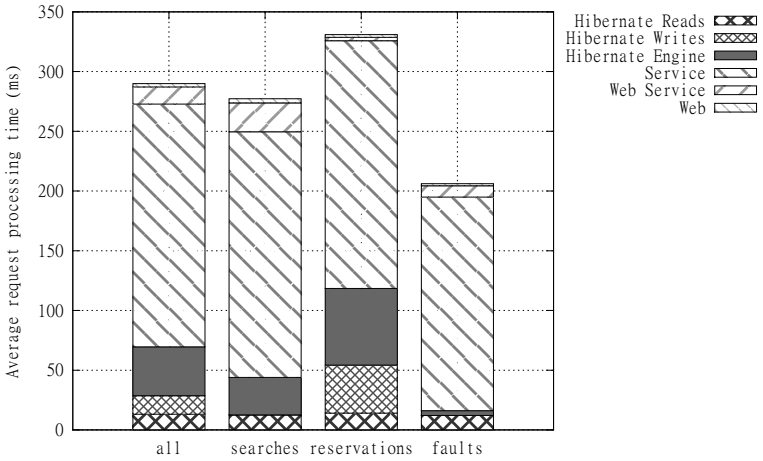


**Fig. 5.** Request processing breakdown for different request types

occurrences of records where the hibernate time was larger than the service time (a physical impossibility).

Since the results were not satisfactory, two new monitor approaches were implemented: *Event* and *Layer*. The Event monitor records one data record to the log for each interception point (just like Perf4J), producing a log file size proportional to the number of accessed objects, and data is written to the log file immediately after each interception. The Layer monitor keeps totals in memory and writes them to file only once per request, at the end of the request processing.

Both *Event* and *Layer* had a lower overhead when compared to *Perf4J*. However only *Layer* was capable of wrapping hibernate objects - Session, Transaction, etc - and correctly handling the nesting of calls between them. This difference is important as Layer monitor without Hibernate wrapping (*Layer -Hwrap* column) shows. It does not capture the Hibernate Engine slice, just like *Event* monitor, and a large slice of Hibernate time is lost. For this greater accuracy, the *Layer* monitor with Hibernate object wrapping was chosen as the final reference monitor that was used for all other experiments.

### 4.5 Request Types

In this experiment, request types are filtered and analyzed separately. Table 3 and Figure 5 present the results.

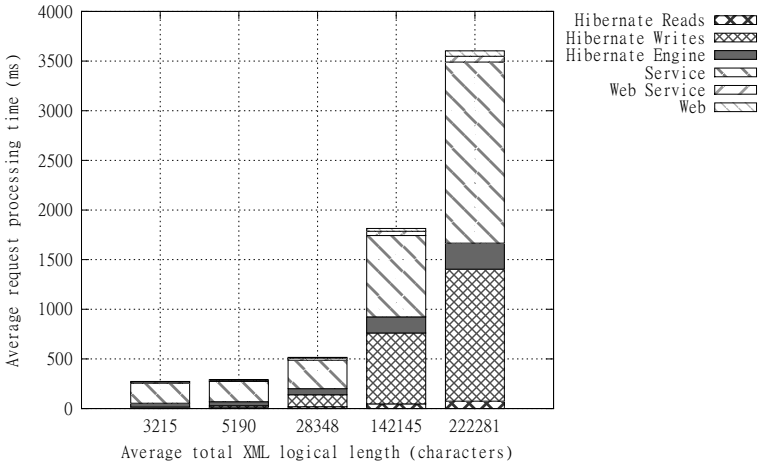


**Fig. 6.** Request processing breakdown for increasing SOAP size

**Table 4.** Request processing breakdown for increasing SOAP size, in percentages

| Avg. XML len. | Web | WS | Service | Hib Eng | Hib W | Hib R |
|---|---|---|---|---|---|---|
| 3215 | 0.96 | 5.26 | 73.93 | 14.33 | 0.80 | 4.73 |
| 5190 | 0.98 | 4.94 | 70.07 | 14.13 | 5.35 | 4.52 |
| 28348 | 1.53 | 3.93 | 55.69 | 11.78 | 23.57 | 3.51 |
| 142145 | 1.60 | 2.35 | 45.16 | 8.95 | 39.35 | 2.59 |
| 222281 | 1.50 | 1.64 | 50.62 | 7.28 | 36.88 | 2.08 |

Searches are read-only, reservations are read-write. Faults were mostly produced by invalid input, so no data was written. Notice how Hibernate Writes slice are empty on searches and faults. The framework handling of failed transactions is efficient because significant time savings are achieved when there is a database rollback.

### 4.6 Web Service Message Size

In this experiment, the SOAP message size is increased by making flight reservation requests with more passengers. Figure 6 and Table 4 present a comparison of the different workloads with increasing average XML length. The dominant slices are still Service and Hibernate. The impact of request time is very significant, above linear progression. Figure 7 shows the detail only for the Web and SOAP slices. The XML processing behavior is also increasing above linear progression.

Increasing XML size has less impact than initially predicted, providing evidence that XML parsers have been greatly optimized since the early versions where the performance degradation was more significant [13]. However, there are still practical limits for the message sizes: for messages above 150,000 characters (roughly 150 KiB assuming UTF-8 encoding) the server starts to fail with java.lang.OutOfMemoryError due to lack of Java heap space. This explains why the percentage of time spent in the service layer (c.f. ‘Service’ column in Table 4) actually decreases with increasing XML length.

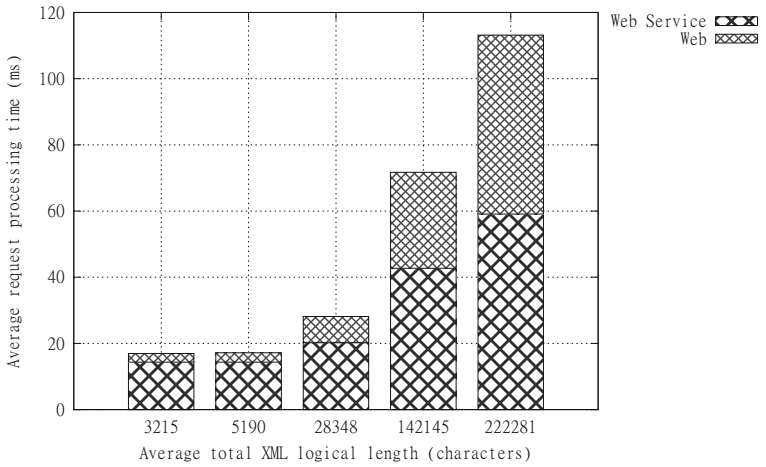


**Fig. 7.** Web and Web Service layers detail of request processing breakdown with increasing SOAP size

**Table 5.** Request processing breakdown for different cache settings, in percentages

| Configuration | Web | WS | Service | Hib Eng | Hib W | Hib R |
|---|---|---|---|---|---|---|
| Local DB | 0.98 | 4.94 | 70.07 | 14.13 | 5.35 | 4.52 |
| w r-only cache | 0.95 | 5.28 | 70.46 | 13.63 | 4.81 | 4.88 |
| w r-w cache | 0.91 | 5.05 | 65.32 | 13.40 | 4.67 | 10.65 |
| 100 Mbit LAN DB | 0.72 | 4.06 | 65.20 | 16.01 | 8.96 | 5.06 |
| w r-only cache | 0.75 | 4.42 | 65.74 | 14.93 | 8.61 | 5.54 |
| w r-w cache | 0.68 | 4.19 | 62.33 | 14.76 | 8.16 | 9.88 |
| 10 Mbit LAN DB | 0.28 | 1.88 | 78.50 | 6.83 | 10.64 | 1.88 |
| w r-only cache | 0.32 | 2.53 | 77.66 | 6.55 | 10.86 | 2.08 |
| w r-w cache | 0.25 | 1.90 | 77.28 | 6.80 | 10.26 | 3.51 |

### 4.7 Hibernate Second-Level Cache

The goal of this experiment was to measure the improvement of performance by using the out-of-the-box Hibernate second-level caching [15], EHCache (Easy Hibernate Cache). The first-level cache is turned on by default and is managed at the Hibernate Session object. Since each request has its own Session, the cache is not shared between them. The second-level cache is managed at the Session Factory object and allows sharing between sessions.

When running Tomcat and MySQL in the same machine, using the second level cache actually did *not* improve performance (c.f. first 3 rows of Table 5). The read-only cache has negligible effect (c.f. next 3 rows). The read-write cache actually decreases performance (c.f. last 3 rows).

When running Tomcat in machine A and MySQL in machine B, connected by a 100 Mbit LAN, the results were only marginally worse, despite the network communication.

Only when running Tomcat in machine A and MySQL in machine B, connected by a

**Table 6.** Request processing breakdown for increasing concurrent users

| Users | Web | WS | Service | Hib Eng | Hib W | Hib R |
|---|---|---|---|---|---|---|
| 1 | 0.98 | 4.94 | 70.07 | 14.13 | 5.35 | 4.52 |
| 2 | 1.10 | 4.89 | 70.06 | 14.08 | 4.08 | 5.80 |
| 4 | 1.21 | 4.04 | 71.09 | 13.58 | 3.97 | 6.11 |
| 8 | 2.04 | 4.62 | 65.06 | 17.07 | 5.47 | 5.74 |
| 16 | 2.75 | 6.07 | 62.55 | 19.56 | 4.80 | 4.26 |

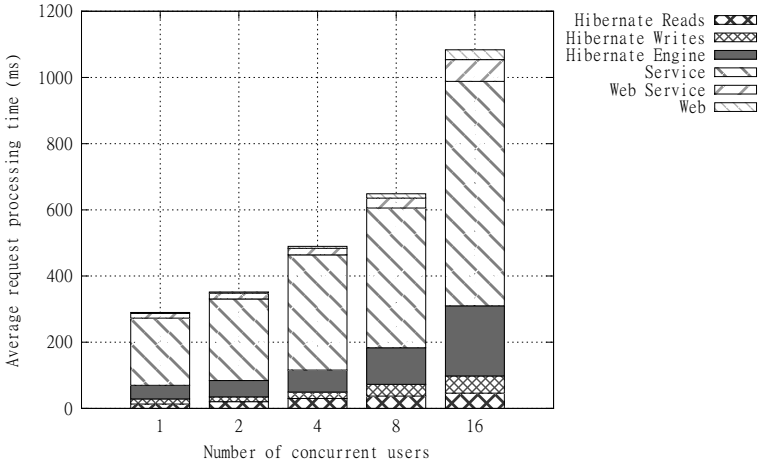


**Fig. 8.** Request processing breakdown for increasing concurrent users

more constricted 10 Mbit LAN, did the read-only cache prove beneficial. However, the request processing time for this configuration was approximately 3 times slower than the others.

The best solution for this application is to leave the second-level cache turned off as most caching benefits were achieved with the first-level cache.

### 4.8 Concurrent Users

The performance of an application in a production environment heavily depends on the number of users, making it hard to properly test the implementation in a development environment where a single user is available. In this experiment several virtual users were running at the same time. Table 6 and Figure 8 present the results.

The server scales reasonably well for the tested number of users. The request processing time stays in the same order of magnitude for a ten-fold increase in load (from 1 user to more than 10, it stays near the 1 second range).

However, there is a problem: the number of Application Exceptions stays the same (as expected in a simulated workload) but the number of System Errors steadily increases, from 0% for 1 user, to 30% for 16 users. This is caused by Hibernate optimistic cache [15] approach that throws org.hibernate.StaleObjectStateException when it detected concurrent modifications of the same objects. This happens not only for entity

**Table 7.** Log level average processing time and average functional log size

| Log level | Time (ms) | Log size (bytes) |
|---|---|---|
| Off | 332.52 | 0 |
| Fatal | 332.10 | 0 |
| Error | 331.69 | 1792 |
| Warn | 333.70 | 1792 |
| Info | 332.91 | 13978 |
| Debug | 4431.41 | 296059571 |
| Trace | 37430.76 | 2029488189 |

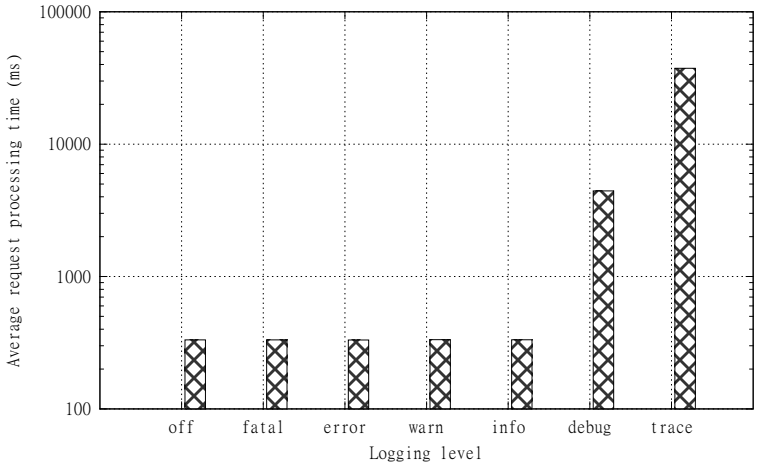


**Fig. 9.** Request processing times for log level settings. The y axis is in logarithmic scale.

data modifications, but also for relationship modifications. The impact of this issue is magnified because the STEP Framework cookbook[15] advocates the use of a "Domain Root" object that connects to all the main domain entities. This guideline has a measurable impact on the scalability of STEP applications and should be reconsidered in future versions.

### 4.9 Logging Cost

Log libraries are very important for server-side applications as a debug and diagnostics tool. The STEP Framework and the libraries it uses rely on Apache Log4J[16] to log program messages. In this experiment, the functional log level was changed from no messages ("off") up to the most detailed level ("trace"). Table 7 and Figure 9 present the results.

The cost of logging beyond "info" level is enormous, making the "debug" and "trace" levels impractical for production environments.

[15] Cookbooks available at `http://stepframework.sourceforge.net/`
[16] `http://logging.apache.org/log4j/`

Additional detail levels could help alleviate this problem, as well has selecting partial output only from some of the layers and not all of them, or activating them for a subset of requests (e.g. requests from a specific user).

## 5 Conclusions

This chapter presented the performance assessment of a representative Web Service developed using the STEP Framework. Performance monitoring is much harder than first expected. Also, assembling a tool chain to collect, process, and visualize the data is an extensive work. But the benefits of having it in place are greatly beneficial for development, especially in an open-source, academic learning environment.

The detailed description of the performance analysis process provides insight to how similar techniques can be used in other frameworks, and how to avoid some of the pitfalls, in particular, regarding monitor implementation and how measurements should always be interpreted with regard for the bias introduced by the measurement process itself.

The presented experiment findings – time slice breakdown, monitors comparison, request types, SOAP size, caching, concurrent users, and logging – are illustrative of the framework's new capabilities and of how they can be used by learning developers make more informed decisions that help give better performance to their Web Services.

**Acknowledgements.** Miguel L. Pardal and Joana Paulo Pardal are supported by PhD fellowships from the Portuguese Foundation for Science and Technology FCT (SFRH/BD/45289/2008 and SFRH/BD/30791/2006).

The authors wish to thank Prof. Paulo Jorge Pires Ferreira for his insightful review of an earlier manuscript.

## References

1. Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., Stafford, R.: Patterns of Enterprise Application Architecture. Addison Wesley (2002)
2. Jain, R.: The Art of Computer Systems Performance Analysis - Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley (1991)
3. Menascé, D.A., Almeida, V.A.F., Dowdy, L.W.: Performance by Design - Computer Capacity Planning by Example. Prentice Hall (2004)
4. Pardal, M., Fernandes, S., Martins, J., Pardal, J.P.: Customizing Web Services with Extensions in the STEP framework. Int'l Journal of Web Services Practices 3(1) (2008)
5. Pardal, M.: Core mechanisms for Web Services extensions. In: 3rd Int'l Conf. on Next Generation Web Services Practices (NWeSP). IEEE Computer Society (2007)
6. Leitão, J.C.C., Pardal, M.L.: Smart Web Services: systems integration using policy driven automatic configuration. In: Quintela Varajão, J.E., Cruz-Cunha, M.M., Putnik, G.D., Trigo, A. (eds.) CENTERIS 2010, Part II. CCIS, vol. 110, pp. 446–454. Springer, Heidelberg (2010)
7. Roza, M., Schroders, M., van de Wetering, H.: A high performance visual profiler for games. In: ACM SIGGRAPH Symp. on Video Games (Sandbox 2009), pp. 103–110. ACM, New York (2009)

8. Shankar, K., Lysecky, R.: Non-intrusive dynamic application profiling for multitasked applications. In: 46th Annual Design Automation Conf. (DAC), pp. 130–135. ACM, New York (2009)
9. Pearce, D.J., Webster, M., Berry, R., Kelly, P.H.J.: Profiling with aspectj. Softw. Pract. Exper. 37, 747–777 (2007)
10. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous Java performance evaluation. In: 22nd Annual ACM SIGPLAN Conf. on Object-Oriented Programming Systems and Applications (OOPSLA), pp. 57–76. ACM, New York (2007)
11. Montgomery, D.C., Runger, G.C.: Applied Statistics and Probability for Engineers. Wiley (2010)
12. Pardal, M.L., Pardal, J.P., Marques, J.A.: Improving Web Services performance, one STEP at a time. In: 2nd Int'l Conf. on Cloud Computing and Services Science (CLOSER) (2012)
13. Machado, A., Ferraz, C.: JWSPerf: A performance benchmarking utility with support to multiple web services implementations. In: Int'l Conf. on Internet and Web Applications and Services (ICIW), pp. 159–159 (2006)
14. Boyer, B.: Robust Java benchmarking. IBM Developer Works (2008)
15. Bauer, C., King, G.: Java Persistence with Hibernate. Manning (2006)