Performance assessment of the STEP Framework

Miguel L. Pardal

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

Abstract. This technical report presents a performance study of the STEP Framework, an open-source application framework based on the Java platform that has been used for several years to teach the development of distributed enterprise applications to Computer Science and Engineering undergrad students.

The findings are reported as required for the *'Advanced Distributed Systems'* course, taught by Prof. Paulo J. P. Ferreira, as part of the Doctoral Program in Computer Science and Engineering at Instituto Superior Técnico, Technical University of Lisbon.

1 Introduction

Enterprise applications have many demanding requirements [1], and some of the most important are related to *performance*. To verify if an implementation is performing as expected, runtime data must be collected and analyzed. This data can also be used to compare design and configuration alternatives.

Performance analysis is a challenge [2] [3], that can be especially hard for new developers.

The *Simple, Extensible, and for Teaching Purposes (STEP) Framework*[1] is an open-source, multi-layer, Java-based enterprise application framework for developing Web Applications (Servlet/JSP) and Web Services. Its source code is intended to be small and simple enough to allow any developer to read and understand it thoroughly, learning how the architectural layers are implemented in practice.

The framework has been used for several years in 'Software Engineering' and 'Distributed Systems' courses lectured at Instituto Superior Técnico (IST) of the Technical University of Lisbon (UTL), to teach Computer Science and Engineering undergrads how to develop Web Services (WS) with enterprise-like requirements.

Before the work described in this report, the STEP framework did not provide means to collect runtime 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 make informed decisions during development and deployment.

This report continues with a brief overview of the STEP framework architecture before presenting the *performance assessment study* including the new tools and the results of the conducted experiments.

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

2 STEP Framework overview

2.1 Architecture

The STEP Framework defines a typical layered architecture [1], to separate the implementation of different concerns. The main layers are Domain and Service. There are also Presentation and View layers, as depicted in Figure 1.

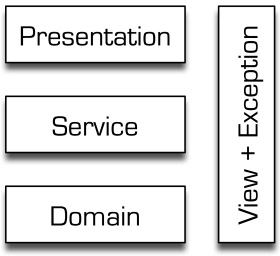

Fig. 1. Architectural layers of a STEP Framework application.

The *Domain* layer is where an object-oriented solution for the application functional requirements are implemented. Domain objects are persisted to a database using object-relational mapping.

The *Service* layer provides access to the application's functionalities through service objects, that control access to 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 (including exceptions) for service objects.

The *Presentation* layer is responsible for user interaction through a Web interface, implemented with servlets and Java Server Pages (JSP). There is also a Web Service layer that provides remote access to services.

STEP relies on other open-source libraries to implement some of its layers, namely: Hibernate[2] for the Domain persistence, JAX-B[3] for the View layer, JAX-WS[4] for the Web Services layer, and Stripes[5] for the Web layer.

STEP also supports Extensions [4], a mechanism for intercepting the Service and Web Service layers that eases the implementation of cross-cutting concerns.

2.2 Request processing

A request processing sequence for a STEP Web Service is represented in Figure 2.

[2] `http://www.hibernate.org/`
[3] `https://jaxb.dev.java.net/`
[4] `https://jax-ws.dev.java.net/`
[5] `http://www.stripesframework.org/`

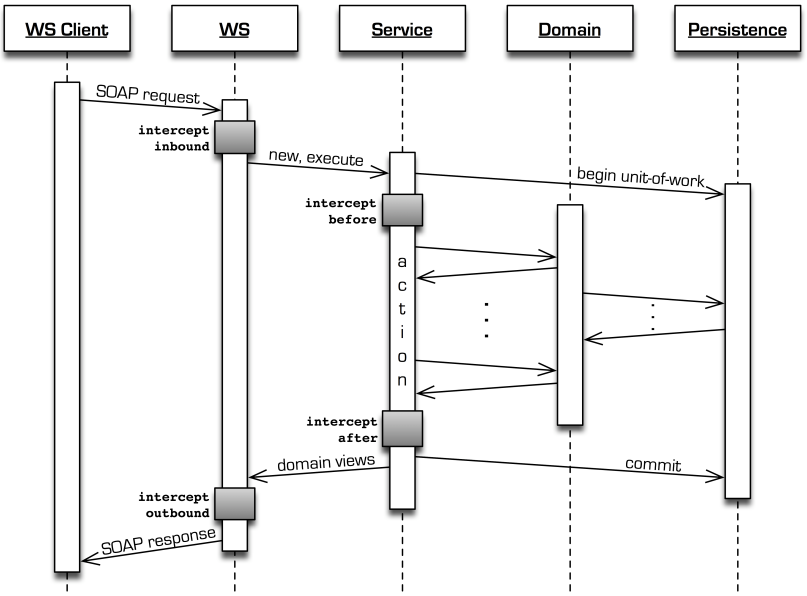

Fig. 2. Sequence diagram of STEP Web Service.

A request begins in the client application that sends a SOAP envelope in an HTTP request to the server. 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 Web Service layer parses the SOAP envelope. The payload is deserialized from XML to Java objects.

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 data persistence library maps entities and relationships to database tables and vice-versa using code annotations. SQL queries are generated and executed automatically.

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 are created and returned to the Web Service 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 assessment

The goal of the performance assessment of the STEP Framework is to collect runtime data, to analyze it, and to test performance improvement hypothesis. The metric used to measure performance is *request processing time*.

The performance of Java programs is affected by application, input, VM, GC, heap size, and underlying operating system. All these factors produce *random errors* in measurements that are unpredictable, non-deterministic, and unbiased [5].

One of the main sources for non-determinism is the JIT compilation process [6], because it is driven by timer-based method execution sampling. Different executions take different samples and can make different compilation choices.

3.1 Dealing with measurement errors

To quantify the random errors in measurement, the program runs had to be repeated several times. The presented values were computed using the *mean* of the samples with a confidence interval (margin of error) computed with a confidence level of either 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 [7].

Only changes in values greater than the error margin can be considered *statistically relevant* changes and not the effect of random errors.

3.2 Hardware and software platform

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

Machine A had a Quad-core[6] CPU running at 2.50 GHz, with 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, Sun 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 had a Dual-core[7] CPU running at 2,53 GHz, with 3 GB of RAM, and 500 GiB of hard disk storage. It ran the same software versions as machine A.

The machines were connected either by a 100 Megabit LAN[8], or by a 10 Megabit LAN[9].

The machines were configured to either disable or postpone all system maintenance activities (file system indexing, software updates, screen savers, etc). The SysInternals[10] tools AutoRuns and ProcMon were used to disable and assert the deactivation, respectively. Power management settings were set for maximum performance.

[6] Intel Core 2 Quad CPU Q8300.

[7] Intel Core 2 Duo CPU P9500.

[8] The actual bandwidth measured was 94,402+-0,157 Mbit/s with 95% confidence

[9] The actual bandwidth measured was 7,312+-0,262 Mbit/s with 95% confidence

[10] `http://technet.microsoft.com/en-us/sysinternals/default.aspx`

The parameters for JVM heap size and garbage collection were kept at default values.

The measurements were taken for the application's steady-state performance (and not for start-up performance). GC and object finalization were not forced for the test runs, because they were considered as part of the steady-state server workload [8].

Unless stated otherwise, all presented results in this report were produced by the Web Service running in machine A.

3.3 Target 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.

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 all of 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 IWS Web Service benchmark [9], making Flight WS a typical Web Service. All operations were implemented following the guidelines in the STEP Framework "cookbook", making Flight WS a typical STEP Web Service. If both these assumptions hold true, conclusions drawn for Flight WS can be applied to similar Web Services.

3.4 Performance analysis process

The performance analysis process encompasses all activities necessary to generate, collect, and analyze performance data. Figure 3 presents the data-flow diagram.

Each of the activities is performed by a specific tool: Domain Data Generator, Load Generator, Load Executor, Monitor, Analyzer, and Report Generator.

A Monitor should have low overhead [6] on the application, so intensive computations were done, whenever possible, before or after runtime i.e. by the Load Generator or by the Analyzer.

3.5 Domain data generator

The Domain Data Generator tool populates the Flight WS database with realistic data, both in values and in size.

The data population programs were programmed using scripts[11] to parse data files and to access the database.

The airport list was populated with all of the world's airport names, locations, and codes, as of April 2008, obtained by Wiebe Cazemier[12]. The chosen airline was British

[11] All scripts mentioned in this paper were implemented using the Groovy language, available at `http://groovy.codehaus.org/`

[12] `http://www.world-airport-codes.com/`

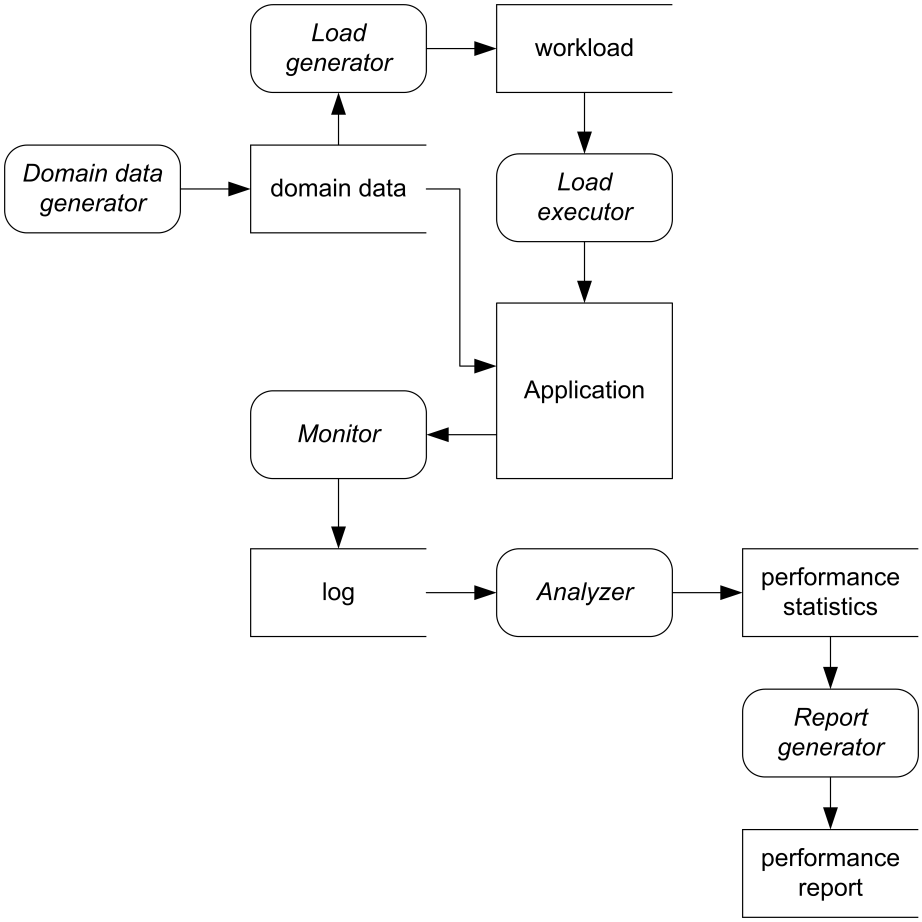

Fig. 3. Performance tool chain data flow diagram.

Airways (BA). The airplane records were generated using the BA fleet information available on Wikipedia on April 2010. The flights were generated by making random picks (with uniform distribution), of airports and airplanes. For each picked pair of origin and destination, a set with a random number of flights was generated, so that search flight requests would could have more than one result. The system was loaded with 15 days worth of flights assuming an 80% use rate in the fleet. The total was 2820 flight records.

The random numbers were generated using the default Java pseudo-random number generator and the MySQL rand() function.

The initial database population was kept the same for the duration of the study. Whenever it was necessary to revert the database back to the initial state, the generation script was run again with the same input data files and the same random number seed.

3.6 Load generator

The Load Generator tool produces files with serialized request objects.

The generated user sessions use the following template:

1. Search flights from random existing airports;
2. Randomize user think time;
3. Randomize passenger group size;
   - If size is 0, then no reservation request is created;
   - If size is 1, then a single reservation request is created;
   - If size is greater than 1, then a multiple reservation request is created.

The default workload "SizeMedium" uses the following settings: 50 users sessions (pairs of search and create reservation requests), maximum reservation group size of 100 persons, no think time, and 12.5% chance of (simulated) input errors. These values were selected ad hoc.

The user think times are random millisecond values, with an upper bound. The passenger names were generated randomly from a list of names and surnames. All random values were determined by a seed value.

The decision of generating requests off-line was made because the queries to pick data for the requests from the Flight WS database would contend with the service's queries.

3.7 Load executor

The Load Executor tool was programmed to send requests to Flight WS. The script opens an object stream, reads request objects from it, and executes the operations: think (thread sleep), search flights, create single reservation, and create multiple reservations.

The requests are sent to the specified Web Service endpoint. If an error is caught, the output message is logged, and the processing continues.

This tool uses a thread pool of fixed size[13] to run simultaneous virtual users. There is one thread for each simulated user.

The test run procedure is the following:

[13] Implemented using classes in the java.util.concurrent package.

1. Drop database;
2. Create database (using hibernatetool);
3. Populate database;
4. Reset Tomcat server (remove applications and delete temporary files);
5. Start Tomcat server;
6. Apply configuration and source file changes;
7. Compile modified Flight WS application;
8. Deploy Flight WS in Tomcat server;
9. Start virtual user(s);
10. Stop Tomcat server;
11. Collect log files.

This procedure is fully automated, allowing for the execution of multiple test runs in sequence, unattended.

3.8 Monitor

The Monitor[14] is the core component of performance analysis. It is a new component of the STEP framework that, when enabled, collects request processing times for each architectural layer.

Interception The monitor needs to intercept request processing at interesting interception points represented in Figure 2.

The JAX-WS servlet is intercepted using a Servlet Filter component, configured in the web application descriptor (web.xml) file.

The inbound and outbound SOAP messages are intercepted using a JAX-WS Handler, configured in the JAX-WS code generation configuration file (jaxws-server-custom-binding.xml). A similar interception is also achieved using a STEP Web Service Interceptor, configured in a STEP extension configuration (extensions.properties) file.

The services (business logic objects) are intercepted using a STEP Service Interceptor[15], also configured in the STEP extension configuration file.

The persistence loads and stores, for each object, are intercepted using an Hibernate Event Listener, configured in the persistence configuration (hibernate.cfg.xml) file. The persistence engine objects - SessionFactory, Session, Transaction, Criteria - are wrapped with performance monitoring objects implementing the same interfaces.

The monitor API records the timestamps for entries and exits in each layer using the enter(Object tag) and exit(Object tag) methods. Additional data can be recorded as key-value pairs using the context(Object tag, String key, String value) method.

When all the described interceptors are enabled, one Web Service request is intercepted once by the Servlet Filter, JAX-WS Handler, and STEP Web Service Interceptor,

[14] The Monitor described here is the final version. Several alternatives where tested, as discussed in section 3.11.

[15] The existing service interception mechanism of the framework had to be modified. A "finallyAfter" interception point was added to allow the interception at service level even when an error occurred, because the existing "after" interception point would not be called in these situations.

and multiple times by the STEP Service Interceptor (once for each invoked service), and Hibernate Event Listener and Wrappers (once for each load or store access, and once for each engine object method call). These multiple interceptions at the same layer mean that the total time spent inside a layer is the sum of all elapsed times between entries and exits and that nesting must be tracked properly. The elapsed time is computed by subtracting the timestamp of the first entry from the timestamp of the last exit.

Data logging For each monitored request, a single request record (shown in Figure 4) is logged. A record contains the total time spent in each layer, but not the individual entries and exits.

```
thread[17] tag[filter] accTime[1558494253ns] context[]
thread[17] tag[hibernate] accTime[294309343ns] context[]
thread[17] tag[si] accTime[441975142ns] context[...]
thread[17] tag[soap] accTime[1506191620ns] context[...]
thread[17] tag[wsi] accTime[1504073613ns] context[]
```

Fig. 4. Monitor request record. Context key-value pairs are omitted.

A request record contains one line for each used layer. Each layer record contains the following fields: "thread" to identify the thread, "tag" to identify the layer, "accTime" with the accumulated time inside the layer in a given time unit, and "context" with key-value pairs, separated by commas. The request record is ended by a blank line.

The monitor stores the performance data to a thread-specific log file to avoid contention between threads. The request data is logged after the final timestamp is recorded.

At the end of the run, all log files are merged and the "thread" field is added.

3.9 Analyzer

At this stage all samples of execution data, resulting from multiple runs using the same settings, have been collected and await processing.

The Analyzer tool applies a sequence of processing steps to the sample data and outputs statistics. The statistics calculations use the Apache Commons Math[16] library.

The first processing step converts the request records from monitor format (figure 4) to Comma-Separated Values (CSV) format (figure 5), with one request per line. All context key-value pairs are converted to fields. All time units are normalized to milliseconds.

At this stage, the following conditions are verified and warning messages are produced if any of them does not hold true:
Web time $>$Web Service time $>$sum of Service time $>$sum of Hibernate time.

Next there is a filter that can be used to select a subset of records satisfying a specific condition (e.g. request must be flight reservation).

[16] `http://commons.apache.org/math/`

```
filter_time,soap_time,soap_name,soap_request_logical_length,...
1558.494253,1506.19162,searchFlights,204,...
413.706227,406.875026,createMultipleReservations,2254,...
...
```

Fig. 5. Two request records in CSV format. Only the first 4 columns are displayed.

Then, the sample statistics are computed. 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. For each numeric mean field, the margin of error is computed for confidence levels of 90%, 95%, and 99%.

A similar procedure is applied to the virtual user output logs to produce error statistics, from the WS client perspective.

3.10 Report generator

The Report Generator uses the statistical data produced by the Analyzer and uses it to produce custom reports.

The data required for a specific report is fetched in a script, and then visualization tools are used to produce graphical representations.

The gnuplot [10][17] tool was used to produce the plots presented in section 3.11.

The Graphviz[18]tool was used to produce directed graphs, to help verify all configurations.

3.11 Experiments

Now that the performance analysis tool chain was assembled, several experiments were conducted. The results are presented and discussed in this section.

Request time breakdown Table 1 presents the request processing time breakdown. Figure 6 represents the same data graphically.

The largest time slice is Service (70%). This result makes sense not only because the Service slice encompasses all the application-specific logic, but also because it is the slice where the bulk of "other" processing time is accounted for.

The second largest slice is the sum of Hibernates (24%). Hibernate manages the domain objects in the database. The Hibernate engine slice is significant (14%) because it includes the transaction commit, when data is actually written to the database.

The absolute value of roughly 300 milliseconds average processing time is only meaningful for comparing with other results obtained from the same machine.

[17] http://www.gnuplot.info/

[18] http://www.graphviz.org/

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

Table 1. Request processing time breakdown.

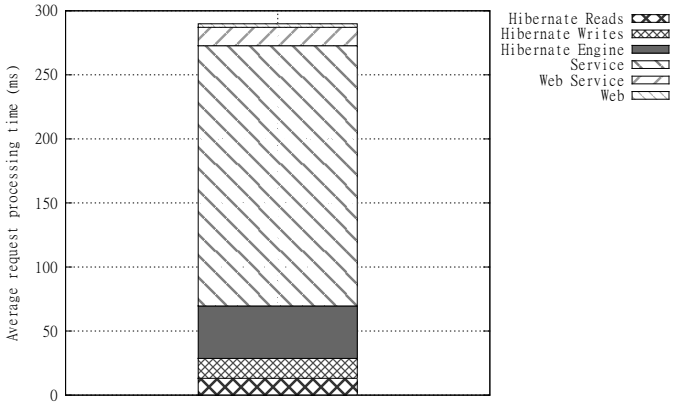

Fig. 6. Request processing time breakdown.

Monitor implementation comparison The final version of the STEP framework performance monitor was presented in section 3.8. However it was not the first implementation. In fact, it was the end-result of several attempts to accurately capture the performance data.

Table 2 and Figure 7 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).

The first choice for monitor was the Perf4J[19] library. Perf4J 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. This allows the 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. Each stop-watch record has a start, time, tag, and (optional) message.

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

Monitor	Web (%)	WS (%)	Service (%)	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

Table 2. Request processing breakdown of different performance monitors.

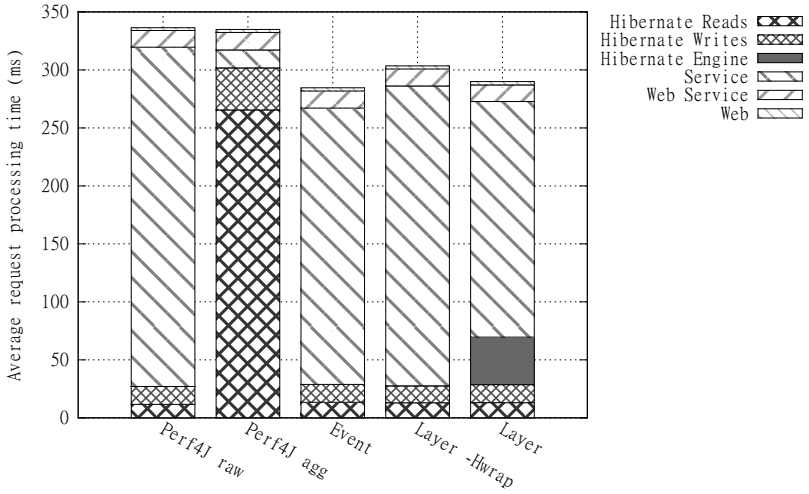

Fig. 7. Request processing breakdown of different performance monitors.

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 (as exemplified in Figure 8).

In the Perf4J monitor with aggregated records (Perf4J agg) the consecutive 0 ms records were combined and the elapsed time was computed using the timestamps. 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 occurrences of records where the hibernate time was larger than the service time, a physical impossibility.

The results were not satisfactory so two new monitor approaches were implemented: Event and Layer. The difference between them is that Event 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. Also, data is written to the log file immediately after each interception. On the Layer monitor, totals are kept in memory and are written only once per request, at the end of the request processing.

```
start[1279756475199] time[12] tag[hibernate.load]
start[1279756475211] time[0] tag[hibernate.load]
start[1279756475211] time[0] tag[hibernate.load]
start[1279756475211] time[0] tag[hibernate.load]
start[1279756475211] time[0] tag[hibernate.load]
...
start[1279756475226] time[0] tag[hibernate.load]
...
```

Fig. 8. Excerpt of Perf4J raw performance log.

Both Event and Layer added two features not supported by Perf4J: performance logging using thread-specific files, and the use of java.lang.System.nanoTime() for added time resolution over java.lang.System.currentTimeMillis()[20].

Both Event and Layer had a lower overhead than 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) shows. It does not capture the Hibernate Engine slice, just like Event monitor, and a large slice of Hibernate time is lost.

So, after this experiment, and for all the mentioned reasons, the Layer monitor with Hibernate object wrapping was chosen as the reference monitor for all other experiments.

Request types In this experiment, request types are filtered and analyzed separately. Table 3 and Figure 9 present the results.

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

Table 3. Request processing breakdown for different request types.

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.

[20] According to the JDK documentation, nanoTime is always guaranteed to be as good as currentTimeMillis. In the used JDK version for Windows, nanoTime in fact provided additional time resolution.

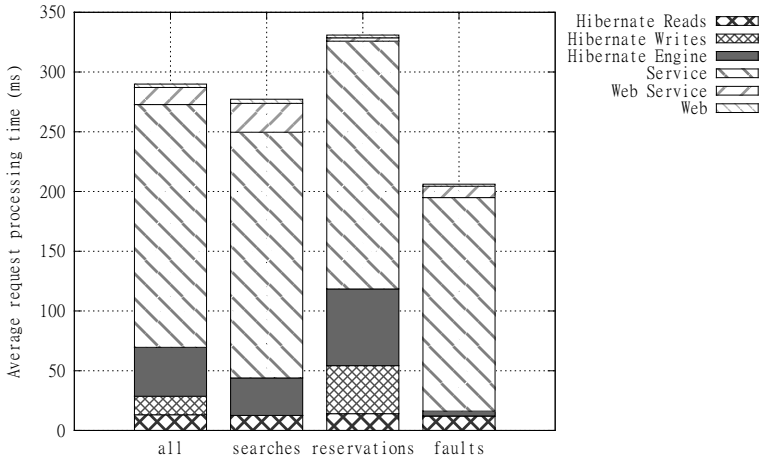

Fig. 9. Request processing breakdown for different request types.

Web Service message size In this experiment, the SOAP message size is increased by making flight reservation requests with more passengers. Figure 10 and Table 4 present a comparison of the different workloads with increasing average XML length.

XML logical length is measured in characters and is the sum of the length of all tag names, attribute names, and text nodes.

Avg. XML length	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

Table 4. Request processing breakdown for increasing SOAP size.

The dominant slices are still Service and Hibernate. The impact of request time is very significant, above linear progression. Figure 11 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 hit was more significant [9].

However, there are still practical limits for the message sizes. Figure 12 shows that 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

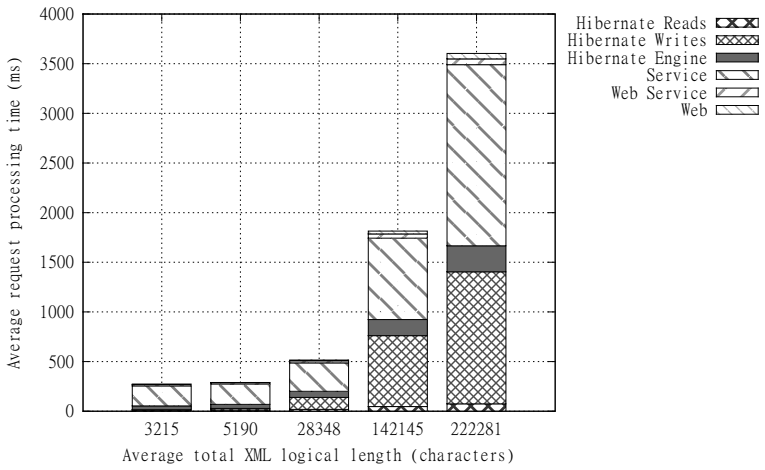

Fig. 10. Request processing breakdown for increasing SOAP size.

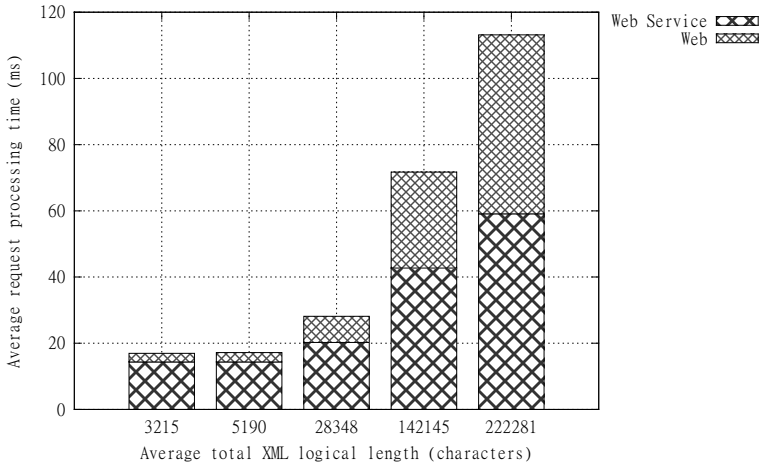

Fig. 11. Web and Web Service layers detail of request processing breakdown with increasing SOAP size.

space. This explains why the percentage of time spent in the service layer (see 'Service' column in Table 4) actually decreases with increasing XML length.

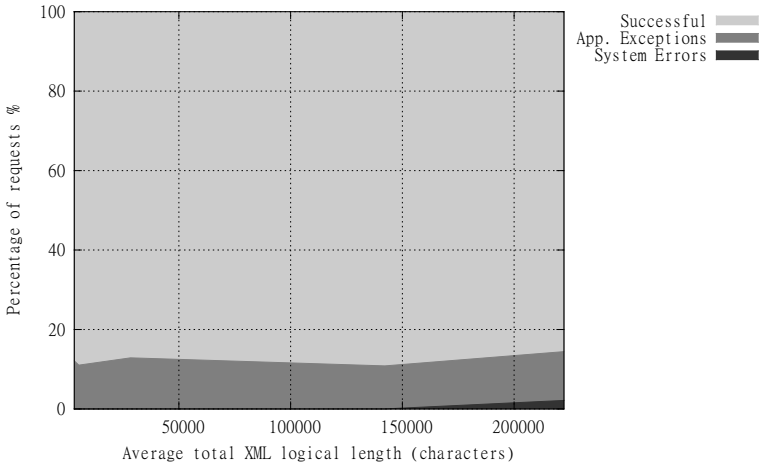

Fig. 12. Outcome of requests for with increasing SOAP size.

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 [11], 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.

The results were disappointing for performance enthusiasts (author included), as can be seen on Table 5 and Figure 13.

When running Tomcat and MySQL in the same machine, using the second level cache actually did *not* improve performance (see first 3 rows of Table 5 and and first 3 columns of Figure 13). The read-only cache has negligible effect (see next 3 rows and 3 columns). The read-write cache actually decreases performance (see last 3 rows and columns).

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

Only when running Tomcat in machine A and MySQL in machine B, connected by a more constricted 10 Megabit LAN, did the read-only cache prove beneficial. However, the request processing time for this configuration was roughly 3 times slower than the others.

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

Table 5. Request processing breakdown for different cache settings.

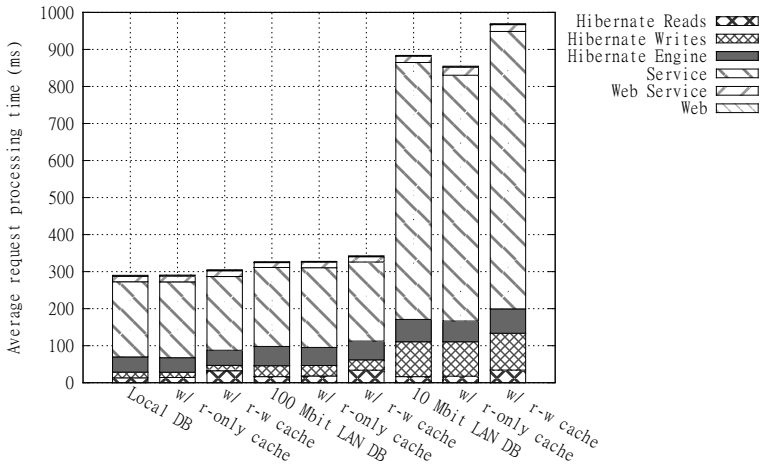

Fig. 13. Request processing breakdown for different cache settings.

The best solution for this application is to leave the second-level cache turned off. Most caching benefits are achieved with the first-level cache. This conclusion might be different for other applications with other data usage patterns.

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 14 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 that is only evidenced by Table 7 and Figure 15.

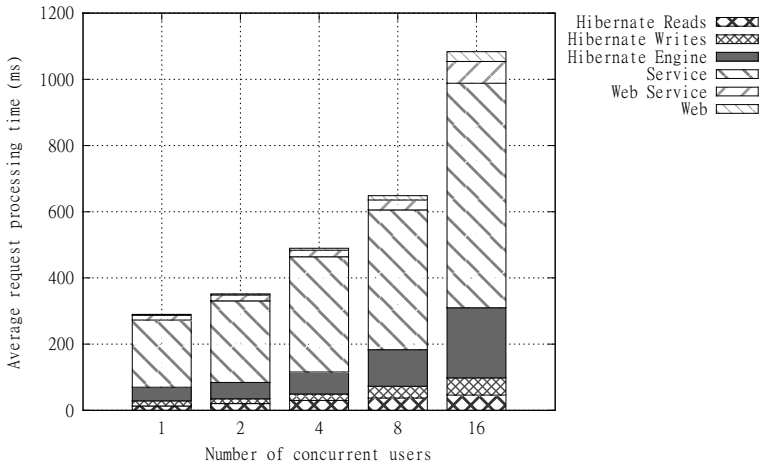

Fig. 14. Request processing breakdown for increasing concurrent users.

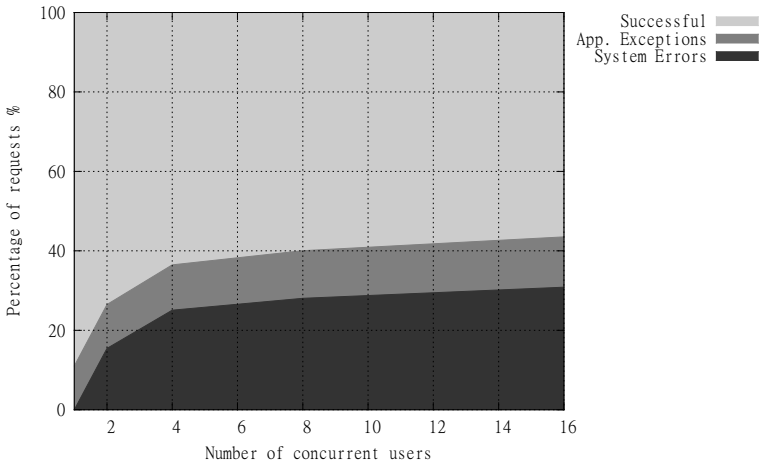

Fig. 15. Outcome of requests for increasing concurrent users.

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

Table 6. Request processing breakdown for increasing concurrent users.

Users	Exceptions (%)	Errors (%)
1	11.04	0.00
2	11.04	15.45
4	11.39	25.06
8	11.98	28.06
16	12.66	30.85

Table 7. Failed requests for increasing concurrent users.

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 [11] approach that throws org.hibernate.StaleObjectStateException when it detected concurrent modifications of the same objects. This happens not only for entity data modifications, but also for relationship modifications. The impact of this issue is magnified because the STEP Framework cookbook 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.

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[21]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 8 and Figure 16 present the results.

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

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).

4 Conclusion

This report presented the performance assessment of a representative Web Service developed using the STEP Framework.

[21] `http://logging.apache.org/log4j/`

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

Table 8. Log level average processing time and average functional log size.

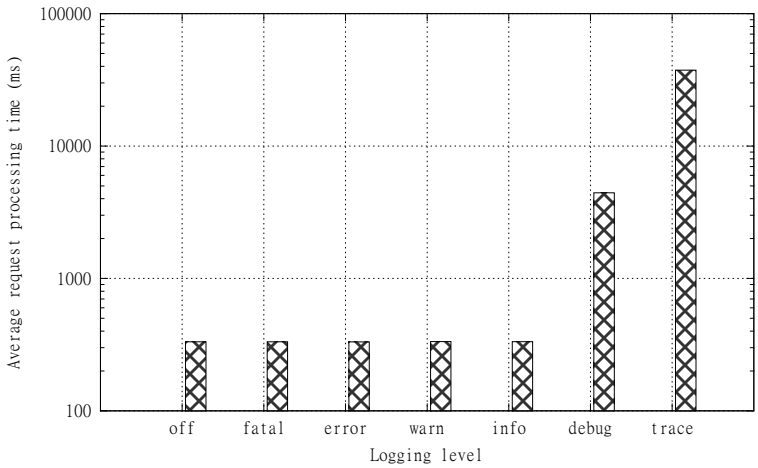

Fig. 16. Request processing times for log level settings. The y axis is in logarithmic scale.

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.

As a specific example, an ongoing MSc thesis will use the new STEP monitor and tools to assess the performance impact of using WS-Policy [12] to achieve automatic configuration of Web Services.

The presented experiment findings - time slice breakdown, monitors, 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 developers to learn more and leave "guesswork" behind. This is the lasting contribution of 'Tough STEP'. The detailed description of the performance analysis process also provides insight to how similar techniques can be used in other frameworks, and some of the pitfalls are stated and explained for others to avoid.

References

1. M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.
2. R. Jain, *The Art of Computer Systems Performance Analysis - Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.
3. D. A. Menasce, V. A. F. Almeida, and L. W. Dowdy, *Performance by Design - Computer Capacity Planning by Example*. Prentice Hall, 2004.
4. M. Pardal, S. Fernandes, J. Martins, and J. P. Pardal, "Customizing web services with extensions in the step framework," *International Journal of Web Services Practices*, vol. 3, June 2008, issue 1. [Online]. Available: http://mflpar.googlepages.com/ijwsp2008-vol3-no1-01.pdf
5. A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA)*. New York, NY, USA: ACM, 2007, pp. 57–76.
6. G. Gousios and D. Spinellis, "Java performance evaluation using external instrumentation," in *Panhellenic Conference on Informatics (PCI)*, 08 2008, pp. 173 –177.
7. D. Griffiths, *Head First Statistics*. O'Reilly, 2008.
8. B. Boyer, "Robust java benchmarking," *IBM Developer Works*, 2008. [Online]. Available: http://www.ibm.com/developerworks/java/library/j-benchmark1.html
9. A. Machado and C. Ferraz, "Jwsperf: A performance benchmarking utility with support to multiple web services implementations," in *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, Feb. 2006, pp. 159 – 159.
10. P. K. Janert, *Gnuplot in Action - Understanding Data with Graphs*. Manning, 2009.
11. C. Bauer and G. King, *Java Persistence with Hibernate*. Manning, 2008.
12. M. L. Pardal and J. C. C. Leitão, "Smart web services: systems integration using policy driven automatic configuration," in *Proceedings of the Conference on Enterprise Information Systems*, 2010.

Acknowledgments

Miguel L. Pardal is supported by a PhD fellowship from the Portuguese Foundation for Science and Technology FCT (SFRH/BD/45289/2008).