Repeated Results Analysis for Middleware Regression Benchmarking

Lubomír Bulej[1,2], Tomáš Kalibera[1], Petr Tůma[1]

[1]Distributed Systems Research Group
Department of Software Engineering
Faculty of Mathematics and Physics
Charles University
Malostranské nám. 25
118 00 Prague, Czech Republic
phone +420-221914267
fax +420-221914323

[2]Institute of Computer Science
Czech Academy of Sciences
Pod Vodárenskou věží 2
182 07 Prague, Czech Republic
phone +420-266053831

lubomir.bulej@mff.cuni.cz
tomas.kalibera@mff.cuni.cz
petr.tuma@mff.cuni.cz

Abstract: The paper outlines the concept of regression benchmarking as a variant of regression testing focused at detecting performance regressions. Applying the regression benchmarking in the area of middleware development, the paper explains how the regression benchmarking differs from middleware benchmarking in general, and shows on real-world examples why the existing benchmarks do not give results sufficient for regression benchmarking. Considering two broad groups of benchmarks based on their complexity, novel techniques are proposed for the repeated analysis of results for the purpose of detecting performance regressions.

Keywords: middleware benchmarking, regression benchmarking, regression testing

1. Introduction

The development and release process of a software system is typically subject to a demand for certain level of quality assurance. One of the approaches to meet this demand is regression testing, where a suite of tests is built into the software system so that it can be regularly tested and potential regressions in its functionality detected and fixed. Regression testing has many potential uses. Applied to the source code of a software system, it verifies the syntactical correctness of the tested code across the range of supported platforms. Applied to the running software system or its parts, it helps guarantee correct functionality of the tested code.

Regression testing is becoming an integral part of the development and release process of communication and application middleware. The complexity of the middleware has led many middleware projects to adopt some form of regression testing, as evidenced by open source middleware projects such as CAROL [2], OpenORB [4], or TAO [8] with its distributed scoreboard [7], which collects results of daily build and test runs on various platforms from across the world. Focusing on functionality, however, the regression testing of middleware tends to neglect the performance aspect of quality assurance, which is typically orthogonal to correct functionality and thus seen as a minor factor in quality assurance. This contrasts with the otherwise common use of middleware performance evaluation to satisfy the obvious need to evaluate and compare performance of numerous implementations of communication and application middleware standards, such as CORBA [17], EJB [10], or RMI [14].

To remedy the existing neglect of the performance aspect in regression testing, we focus on incorporating middleware performance evaluation into regression testing. Our experience from a series of middleware performance evaluation and comparison projects [5][6] shows that systematic benchmarking of middleware can reveal performance bottlenecks and design

problems as well as implementation errors. This leads us to believe that detailed, extensive and repetitive benchmarking can be used for finding performance regressions in middleware, thus improving the overall process of quality assurance. For obvious reasons, we refer to such middleware performance evaluation as regression benchmarking.

In section 2 of the paper, we investigate in more depth the concept of regression benchmarking, explaining why and how it differs from benchmarking in general. Dividing the existing benchmarks into two broad groups based on their complexity, sections 3 and 4 discuss the suitability of the two groups for regression benchmarking and propose techniques for the repeated analysis of results for the purpose of detecting performance regressions. Section 5 concludes the paper.

Throughout the paper, we use TAO [8] and omniORB [12] as real world examples of a complex and mature open-source middleware to illustrate the individual points and proposed techniques.[1] Illustrating the points and techniques on commercial middleware coming from a closed-source vendor would be difficult because the development practices of such a vendor are not public and we would be limited to a user experience with the middleware. Nevertheless, our past middleware performance evaluation and comparison projects have revealed several performance problems in commercial middleware, ranging from minor performance flaws to major scalability issues. These problems would probably have been found had the middleware been subjected to regression benchmarking. From this, we conclude that regression benchmarking also has a valid application in the commercial sphere.

---

[1] The TAO examples use TAO 1.3.1 to 1.3.5 on Pentium M 1.3GHz, 512MB RAM, Linux 2.4.22, GCC 3.3.2. The omniORB examples use omniORB 4.0.3 on Dual Pentium Xeon 2.2GHz, 512MB RAM, Linux 2.4.22, GCC 3.3.2.

2. Regression Benchmarking

Regression benchmarking is a special application of benchmarking for software regression testing. As such it has to be tightly integrated with the development process, fully automated, comprehensive and repetitive. Architecture of an environment for regression benchmarking is outlined in figure 1.

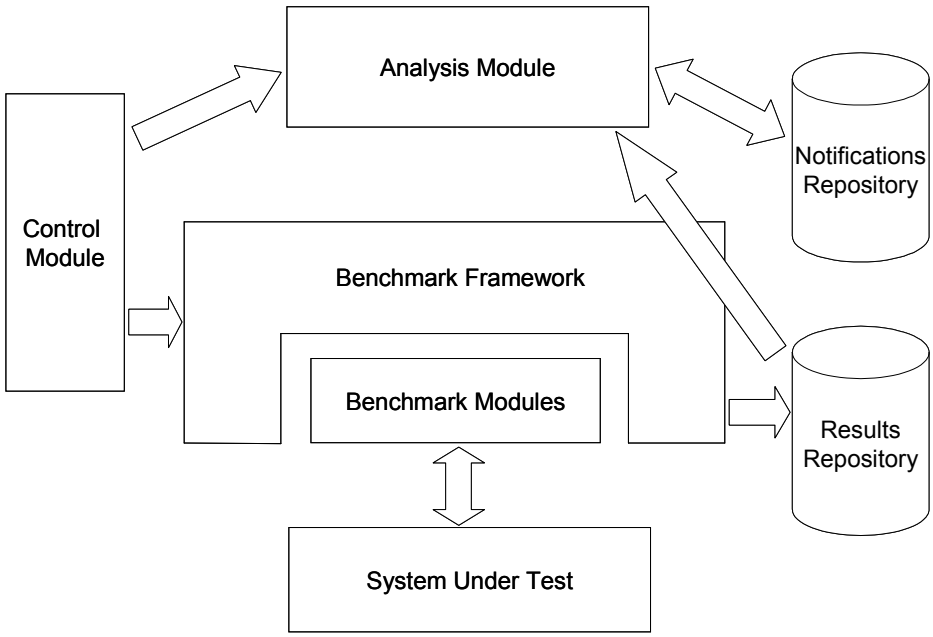

Figure 1: The architecture of an environment for regression benchmarking.

In the outlined architecture, the regression benchmarking is started by the control module, responsible for executing all the configured benchmark modules. To keep the benchmark modules as small and simple as possible, the common functionality of the benchmarks is factored into the benchmark framework that supports the modules. The results collected by the benchmark modules are stored in the results repository, forming a history of results. The analysis module examines the history of results and detects performance regressions. To avoid

repetitive or redundant notifications, the analysis module consults the notification repository for a history of notifications.

The nature of regression benchmarking makes it different from middleware benchmarking in general in the areas of benchmark integration, benchmark automation, result precision and result interpretation.

2.1. Benchmark Integration

The regression benchmarks must be comprehensive in how they cover the functionality provided by the middleware. This is best achieved by integrating the benchmark framework with the middleware so that new benchmark modules can be added alongside new middleware features. The integration minimizes the cost of creating and maintaining benchmark modules.[2]

The integration of the benchmark framework with the middleware has the added benefit of the benchmarks supporting the same platforms as the middleware. Unlike middleware benchmarking in general, the portability of the benchmarks between middleware platforms is less of an issue with regression benchmarking. For a real-world example of the difference, consider the benchmark suite [5], which has 2500 platform-independent and 13500 platform-dependent lines of code to support 12 middleware implementations on 3 operating systems, or the benchmark suite [6], which has 6000 platform-independent and 8000 platform-dependent lines of code to support 4 middleware implementations on 3 operating systems.

---

[2] It is not without interest that the integration fits well with the very similar guidelines for unit testing and acceptance testing from the extreme programming methodology [1].

2.2. Benchmark Automation

The regression benchmarks must be fully automated so that they can run unattended. The requirement of automation concerns not only the execution of the benchmarks, but also the data acquisition and the results analysis. Of these three tasks, automated execution is the simplest, with the existing remote access and scripting mechanisms being well up to the task.

The automated data acquisition must be able to recognize when the regression benchmark outputs stable data as opposed to data distorted during the warm up period of the benchmark. Middleware benchmarking in general either uses long warm up periods or expects the warm up periods to be set by trial and error, neither of which is acceptable for regression benchmarking.

Another problem associated with the automated data acquisition is the need to collect and store large amounts of data without interference with the benchmark. For a real-world example, consider the benchmark suite [6], which generates about 80 megabytes of data in a single run. A rough estimate of an individual observation consisting of the measured operation duration accompanied by annotations such as resource usage data and relevant event lists yields tens of bytes per observation, which in turn yields tens of kilobytes per second for one observation per millisecond.

2.3. Result Precision

The regression benchmarks must detect performance regressions as early as possible. The longer the period between the occurrence and detection of a performance regression, the more

difficult it is to find the source of the regression and the more costly it is to fix the regression. The requirement for early detection implies a need for benchmarks that are so short they can be run daily and so precise they can detect minuscule changes in performance. This is especially a problem for creeping performance degradations, which consist of a sequence of individually negligible changes over a long period of time.

2.4. Result Interpretation

The results of the regression benchmarks are interesting in how they change rather than in what absolute values they have. This means that compared to middleware benchmarking in general, tuning for maximum performance and comparing maximum performance across platforms is less of an issue.

The changes in the results of the regression benchmarks can have many causes, ranging from random fluctuations through effects of inadvertent configuration changes to true performance regressions. The automated interpretation must be able to distinguish these causes reliably to minimize the number of both false positive interpretations and false negative interpretations.

3. Simple Benchmarks

In the paper, we consider the suitability of the existing benchmarks for two broad groups of benchmarks based on their complexity. The group of simple benchmarks covers benchmarks such as [5][6][7], where an isolated feature of the middleware is tested under artificial workload. The intuitive justification for the simple benchmarks is that they provide little space for interference and thus yield precise results with straightforward interpretation.

A real-world example of a simple middleware benchmark is a remote method invocation benchmark that measures the duration of an isolated remote method invocation. The results of such benchmarks in [5][6][7] suggest that individual runs of a simple benchmark typically yield results that differ in units of percents. Using such results for regression benchmarking would imply a need to ignore these differences and only identify differences of tens of percents as performance regressions. Such a precision is clearly too low.

3.1. Minimizing Interference

The difference in the results of individual runs of a simple benchmark can be partially attributed to interference from the operating system, consisting especially of involuntary context switches and device interrupts.

One way of minimizing this interference is keeping the measured operation duration below the period of the interference and thus making the probability of interference during the measured operation reasonably small. In our example, this means measuring the low-level operations that form the remote method invocation, such as the marshaling and unmarshaling operations, data conversion operations and dispatching in various stages of the invocation, rather than the entire remote method invocation. The durations of the low-level operations range from tens to hundreds of microseconds, which is well below the period of the operating system interference, ranging from tens to hundreds of milliseconds.

It is also possible to mitigate the impact of the interference on the results by expressing the results using robust estimators that are not affected by a small number of exceptional observations. In our example, this means using the median of the operation duration rather than the average.

3.2. Collecting Observations

Another reason for the difference in the results of individual runs of a simple benchmark can be an insufficient number of observations collected by each individual run. When estimating the median of the operation duration, we assume the observed durations to be independent identically distributed observations and estimate the median using order statistics. To determine the minimal number of observations necessary to ensure a precise estimate of the median, we employ a quantile precision requirement proposed by Chen and Kelton in [3], which uses a dimensionless maximum proportion confidence half-width instead of the usual maximum absolute or relative confidence half-width. We then determine the required sample size n_p for fixed-sample-size procedure of estimating the p quantile of an independent identically distributed sequence using the formula proposed in [3]:

$$n_p \geq \frac{z^2_{1-\frac{\alpha}{2}} \cdot p \cdot (1-p)}{(\varepsilon')^2}$$

where $z^2_{1-\frac{\alpha}{2}}$ is the $1-\frac{\alpha}{2}$ quantile of the normal distribution, ε' is the maximum proportion half-width of the confidence interval, and $1-\alpha$ is the confidence level.

For a 95% confidence that the median estimator has no more than $\varepsilon' = 0.005$ deviation from the true but unknown median, we need to collect at least $n_p = 38416$ observations. For our experiments, we choose $n_p = 65536$, for which the confidence level borders with 99%.[3]

3.3. Comparing Results

Even after minimizing the interference and collecting the necessary number of observations, the individual runs of a simple benchmark yield different results. A real-world example of a simple middleware benchmark that minimizes the interference by measuring the duration of marshalling as a low-level operation and uses 65536 observations to estimate the median of the operation duration is shown in figure 2.

[3] We have also considered other ways of determining the required number of observations, described in detail in [22].

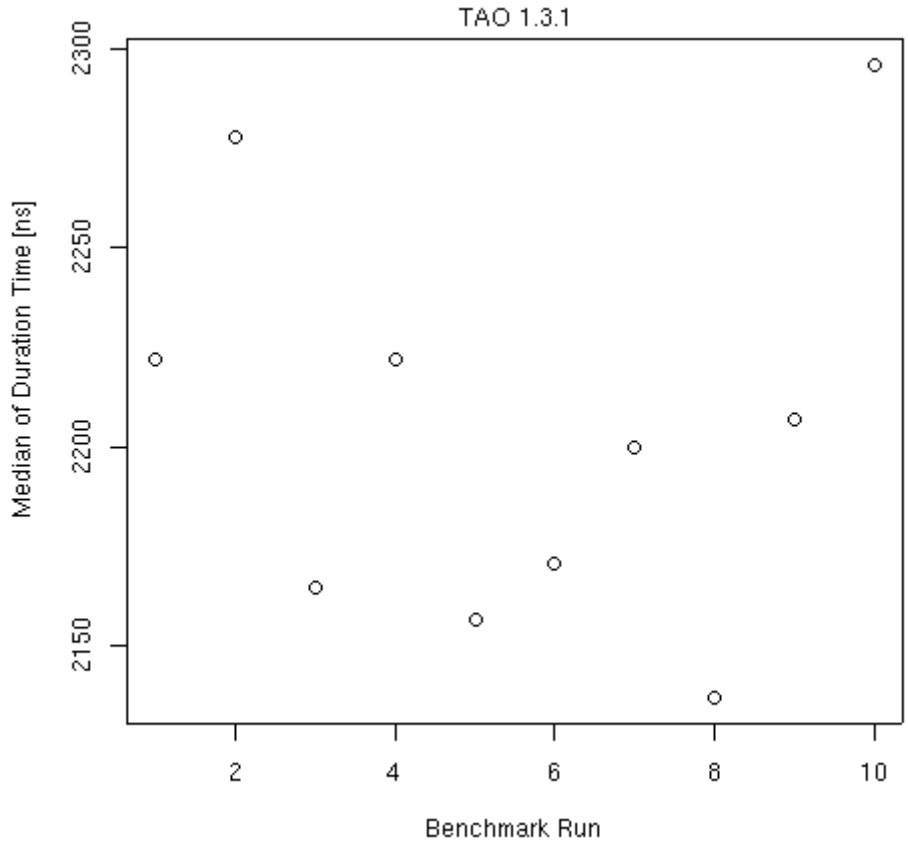

Figure 2: Results of consecutive runs of a benchmark measuring the time to marshal an input array of 1024 CORBA::ULong values on TAO 1.3.1.

The differences in figure 2 suggest that we do not have enough control over the initial state of the system to make the results repeatable across runs, even for a very simple middleware benchmark. This prevents a direct comparison of results from individual runs. Figure 3 illustrates this problem on the results of 10 benchmark runs for two subsequent releases of TAO.

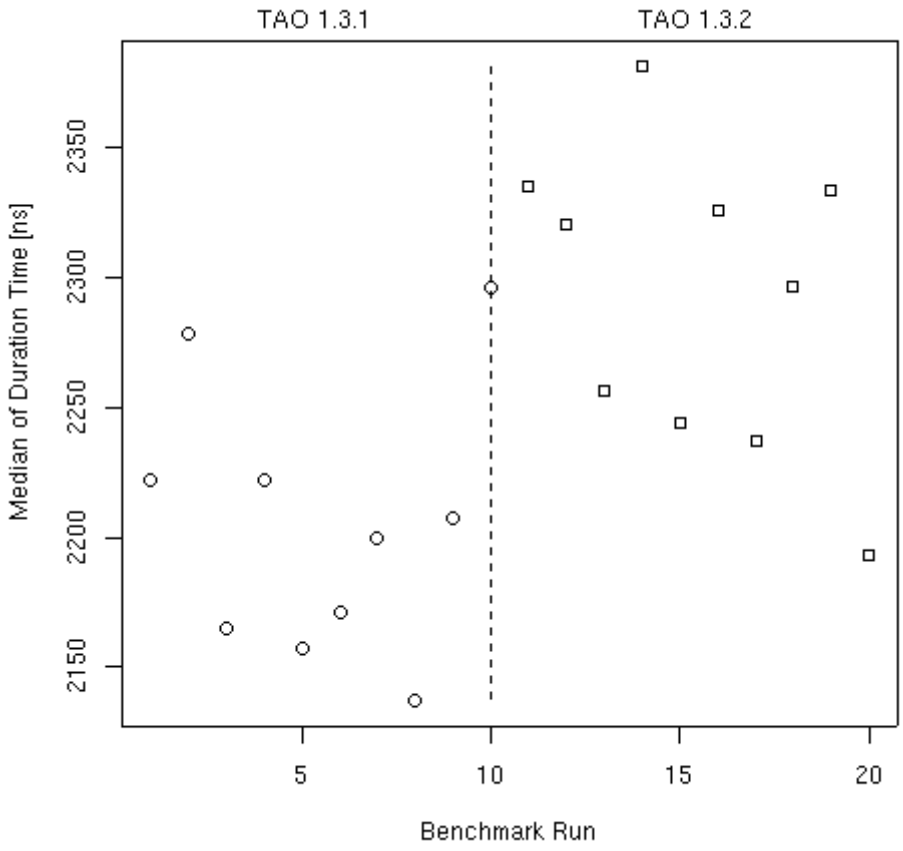

Figure 3: Results of consecutive runs of a benchmark measuring the time to marshal an input array of 1024 CORBA::ULong values on TAO 1.3.1 and 1.3.2.

Although the results in figure 3 indicate that the two subsequent releases of TAO deliver different performance, comparing results for the two releases using only a single run for each release would be potentially incorrect. To compare the results correctly, we have to take into account their random character, and treat the result of an individual benchmark run as an observation of a random variable.

We can assume the results of several consecutive benchmark runs to be a sequence of independent identically distributed observations of a random variable. The assumption of independency and identical distribution can be supported by executing each benchmark run after a system reset, making the initial state of the system for each benchmark run independent

from the initial state for the other runs. Under the assumption of independency and identical distribution, the sets of results from multiple benchmark runs can be compared using the nonparametric statistical tests for comparing samples from two populations, such as Kolmogorov-Smirnov test, Wilcoxon rank sum test, and Kruskal-Wallis test.

Given the generally more pessimistic nature of nonparametric statistical tests when compared to parametric statistical tests, we find it useful to also assume the results of several consecutive benchmark runs to have a normal distribution and apply the parametric statistical tests. The assumption of normal distribution can be tested using Shapiro-Wilk test for normality either directly on the results of consecutive benchmark runs or after applying a normalizing transformation, such as logarithm, reciprocal or reciprocal square root. Samples from two populations with normal distribution can be compared using the unmatched two-sample t-test. Generally, the t-test requires the two samples to have the same variance, but it is also fairly robust against the inequality of variances if the sample sizes are equal, which is our case.

The results of the technique applied to a real-world example are illustrated in figure 4. The example evaluates the development progress of the marshalling mechanism in TAO over five releases from TAO 1.3.1 to TAO 1.3.5, separated by dashed vertical lines in the figure. At the significance level of 0.05, the technique detects changes between TAO versions 1.3.1 and 1.3.2 and TAO versions 1.3.3 and 1.3.4, marked by bold vertical lines. The differences between other releases are not considered significant. The p-values of the results of several nonparametric and parametric statistical tests are tabulated in figure 5.

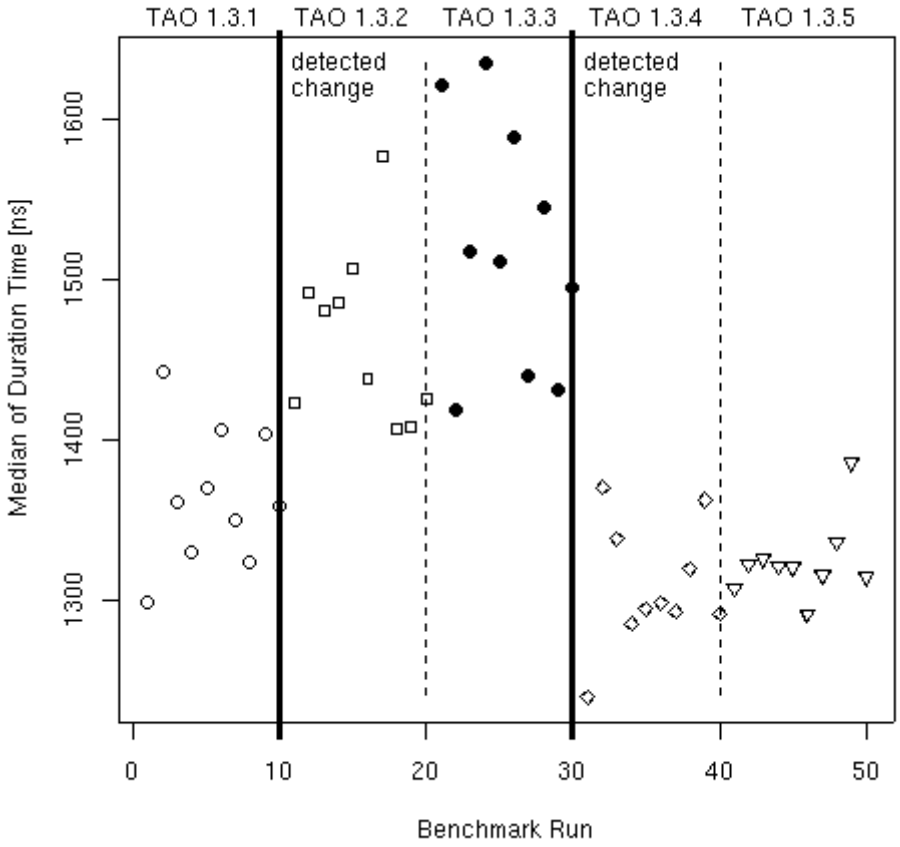

Figure 4: Results of consecutive runs of a benchmark measuring the time to marshal an input array of 1024 CORBA::Octet values on TAO 1.3.1 to 1.3.5.

Results of statistical tests (p-values)	Compared TAO versions			
	1.3.1/1.3.2	1.3.2/1.3.3	1.3.3/1.3.4	1.3.4/1.3.5
Nonparametric (raw data only)				
Kolmogorov-Smirnov	**0.003323**	0.167821	**0.000011**	0.164079
Wilcoxon	**0.000877**	0.052426	**0.000011**	0.256660
Kruskal-Wallis	**0.000765**	0.049366	**0.000157**	0.241145
Parametric (raw and transformed data)				
t-test	**0.000272**	0.078836	**0.000003**	0.328786
t-test, log(x)	**0.000225**	0.079531	**0.000001**	0.318049
t-test, 1/x	**0.000196**	0.080375	**0.000000**	0.307710
t-test, 1/sqrt(x)	**0.000209**	0.079935	**0.000001**	0.312829

Figure 5: Results of nonparametric and parametric statistical tests for consecutive runs of a benchmark measuring the time to marshal an input array of 1024 CORBA::Octet values on TAO 1.3.1 to TAO 1.3.5.

4. Complex Benchmarks

In the paper, we consider the suitability of the existing benchmarks for two broad groups of benchmarks based on their complexity. The group of complex benchmarks covers benchmarks such as [9][18][20], where a set of features of the middleware is tested under real-world workload. The intuitive justification for the complex benchmarks is that they provide results directly applicable to real-world applications.

Complex middleware benchmarks are indispensable because they exercise multiple functions of the middleware concurrently and therefore provide room for effects of complex interactions among the functions to influence the results. Unfortunately, complex benchmarks are more expensive to run than the simple benchmarks in terms of both the cost of the hardware and software setup and the time to run the benchmark. The results of a complex benchmark also have a less straightforward interpretation, especially when expressed as a single value of throughput in a number of operations per second, as in [9][18][20].

These characteristics make the complex benchmarks unsuitable for regression benchmarking. To remedy the situation, we proceed by implementing a simplified version of the TPC-W benchmark [20], which is less expensive to run and collects the duration of individual operations rather than a single value of throughput. The TPC-W benchmark simulates an online bookstore, with a hypothetical architecture in figure 6.

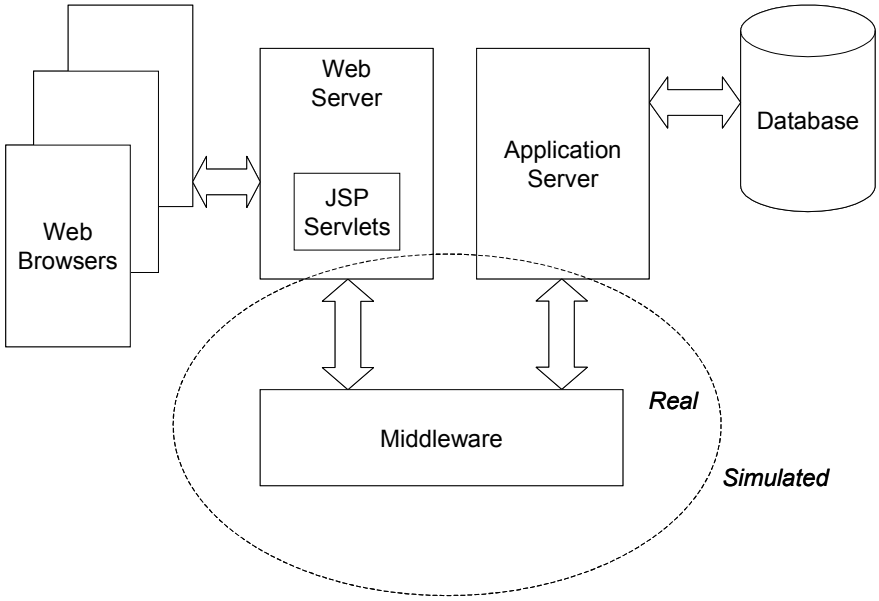

Figure 6: The architecture of an online bookstore for the TPC-W benchmark.

Our simplified version of the TPC-W benchmark reproduces the workload that the TPC-W architecture imposes on the middleware connecting the web server with the application server, without requiring the entire TPC-W architecture to be present. The only interactions that are actually executed and measured are the requests sent by the web server to the application server. The interaction between the web browser and the web server is modeled by sending a sequence of requests to the application server for each request received by the web server. The interaction between the application server and the database is modeled as a delay for each request received by the application server that would normally access the database.

4.1. Comparing Results

Figure 7 shows the duration of one of the frequently executed operations of the simplified TPC-W benchmark, which retrieves a title and author information for a book. The figure contains results obtained by running the same benchmark twice, each time with a different version of

omniORB. One version is the original omniORB, the other version is an artificially damaged omniORB which takes roughly 10% longer to complete a trivial remote method invocation. Ideally, we would want the regression benchmarking to detect the difference.

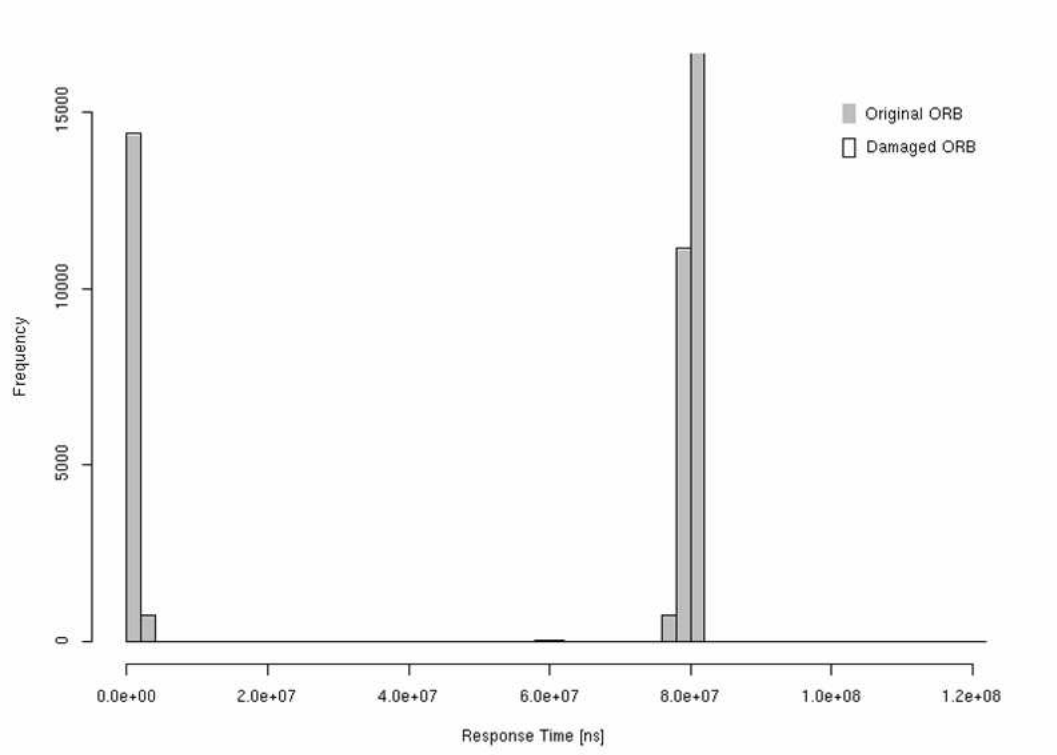

Figure 7: Results of the benchmark measuring the time to retrieve book information on original and damaged omniORB.

As is clear from the figure, the results for the original omniORB and the damaged omniORB are very similar, which means that the methods of comparing averages or medians cannot be used to detect the difference. Indeed, the averages and medians of the results are almost the same for the two versions, as shown in figure 8 for several consecutive runs of the benchmark.

Benchmark version		Response time [μs]			
		Run 1	Run 2	Run 3	Run 4
original omniORB	median	79910	79910	79910	79910
	average	55890	56070	55920	56220
damaged omniORB	median	79910	79910	79910	79910
	average	55850	55920	56330	55540

Figure 8: Medians and averages of the time to retrieve book information.

The reason why the difference between the results of the two versions is difficult to detect is apparent from figure 9, which contains a zoom in of the results. The duration of the operation differs for the cases where the operation takes about 100 microseconds, but the difference is overshadowed by the cases where the operation takes about 80 milliseconds. An analysis of the benchmark would reveal that the longer times occur because of waiting for database object instantiation.

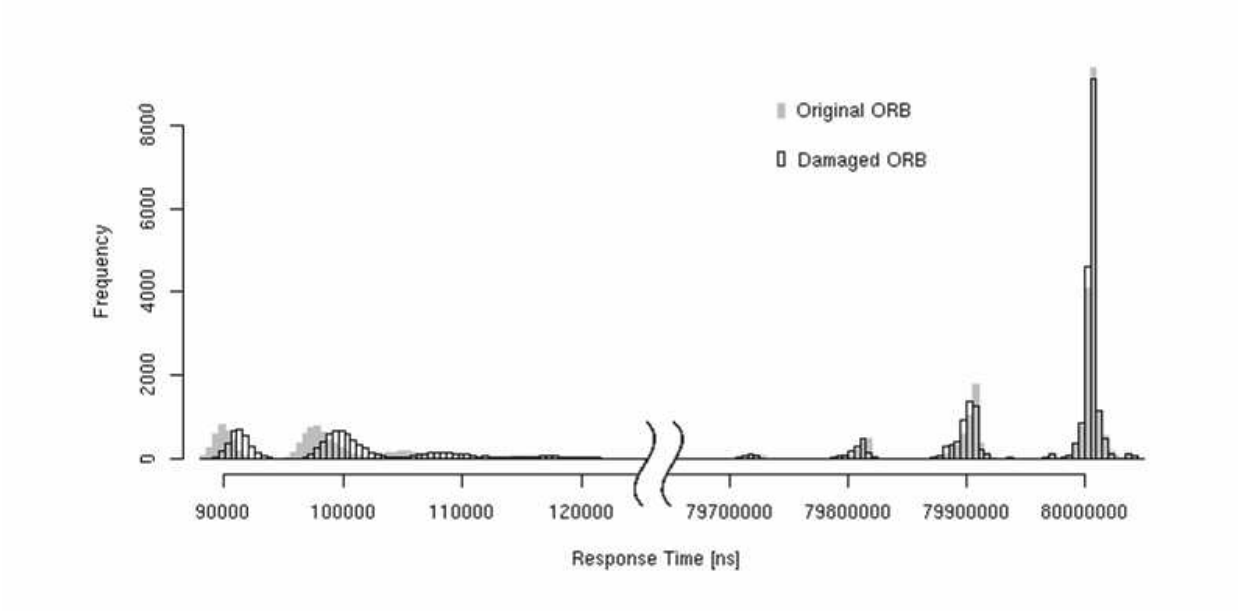

Figure 9: Results of the benchmark measuring the time to retrieve book information on original and damaged omniORB.

4.2. Clusters of Observations

To better understand the results of the two runs, we can intuitively interpret each result as a union of clusters of observations that roughly correspond to specific cases of interactions among the functions of the middleware. An example of a coarse-grained interaction is in figure 7, where the two large clusters correspond to interactions that differ in that one hits while the other misses accessing an object cache. An example of a fine-grained interaction is in figure 10, which shows how the results for three individual operations of the simplified TPC-W benchmark correspond to clusters in the complete set of results.

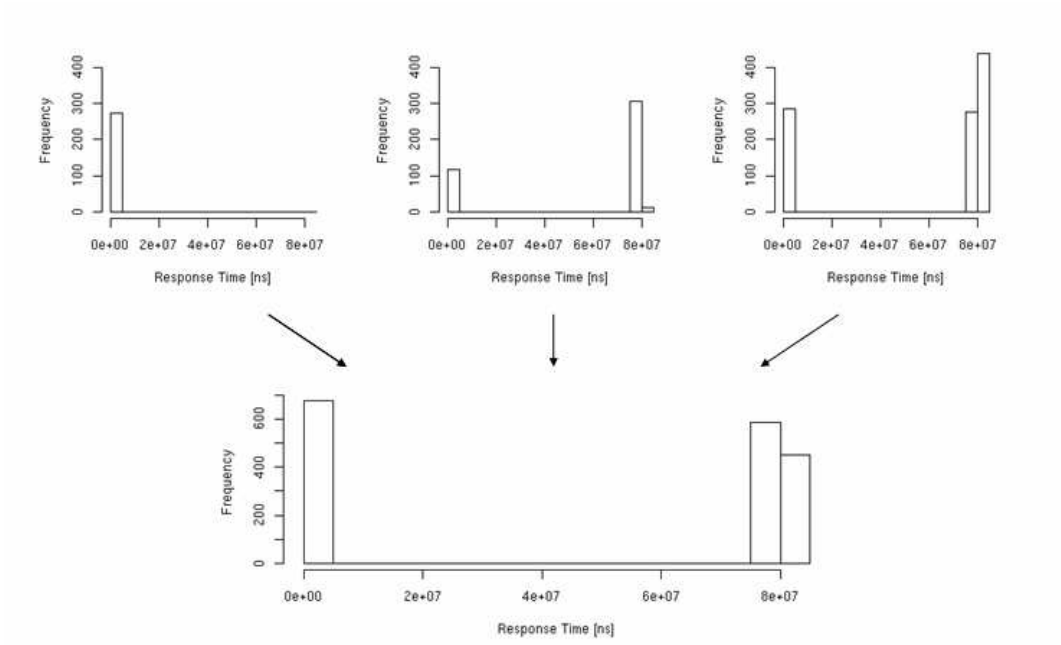

Figure 10: Contribution of three individual operations to the complete results.

While supporting the interpretation of results as a union of clusters of observations, figure 10 also emphasizes that it would be difficult to associate every single cluster with a specific interaction among the functions of the middleware. The degree of insight into the system and the amount of data collected to make such an association is technically prohibitive. Nonetheless,

the differences between the results in figure 7 become visible on individual clusters in figure 9, suggesting that results can be compared cluster-by-cluster.

To compare the results cluster-by-cluster with no information about the association of every cluster with a specific interaction, the clusters need to be identified in the results based on the values of the individual observations only. In our experiments, the best results were obtained when using traditional iterative clustering algorithms that start with a set of initial centers of the clusters and then repeatedly assign data points to the nearest cluster and recompute centers of the clusters. The basic algorithm known as k-means defines the center of a cluster as a mean of the data points in the cluster. The algorithm can find a local minimum of the error measure calculated as a sum of variances of the clusters. Such a minimum corresponds to a centroidal Voronoi configuration, in which each data point is closer to the center of its cluster than to the center of any other cluster [11].

The problem of the k-means algorithm with respect to regression benchmarking is that it requires the initial set of cluster centers to be defined. The centers are often chosen randomly or heuristically. When the centers are badly chosen, the algorithm requires more iteration steps and may find worse solutions. Although improvements of the stability of the clustering results such as bagged clustering [16] exist, the problem of choosing the number of centers remains. In our example, we leave this problem open and select the initial set of cluster centers by hand. The result of the k-means algorithm is in figure 11.

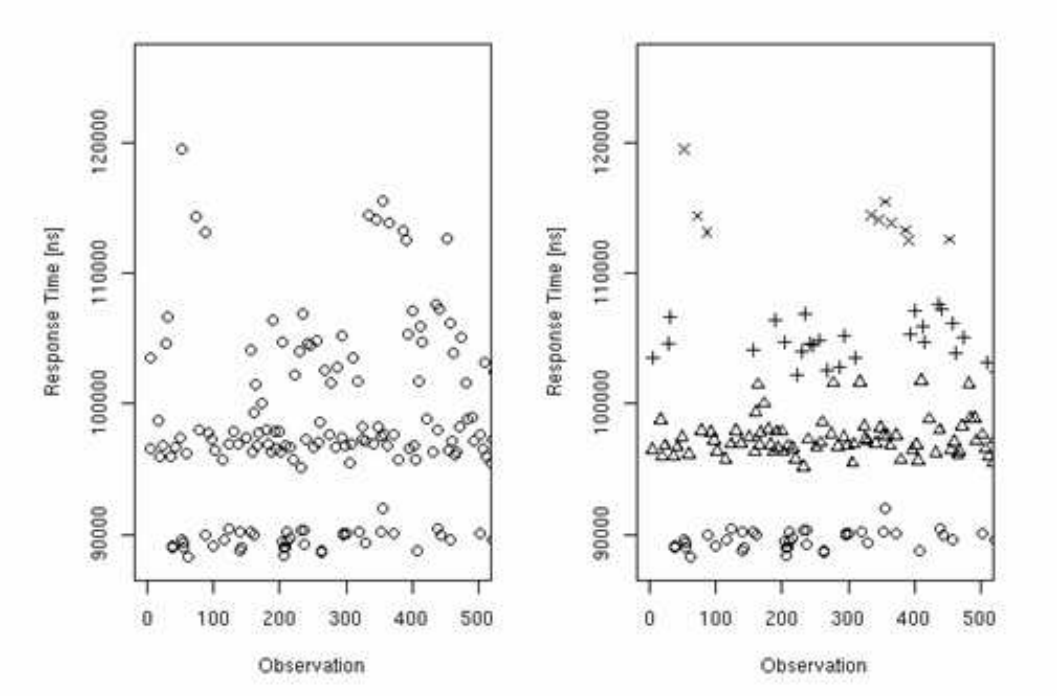

Figure 11: Clusters as identified by the k-means algorithm.

Once the clusters are identified, we can assume the observed durations to be independent identically distributed observations of a random variable and use the technique proposed in section 3. In our example, the technique classifies the identified clusters as having distributions with different means, which is the correct result.

5. Conclusion

We have outlined the idea of regression benchmarking as a variant of regression testing focused at detecting performance regressions. Applying the regression benchmarking in the area of middleware, we explain how the regression benchmarking differs from middleware benchmarking in general in its requirements on benchmark integration, benchmark automation, result precision and result analysis, and show on real-world examples why the existing benchmarks do not give results sufficient for regression benchmarking.

For the group of simple benchmarks, we propose techniques for minimizing interference of the operating system on the benchmark results, collecting the necessary number of observations, and comparing results while taking into account the fact that the results themselves are observations of a random variable. This contrasts with the current practice in middleware benchmarking, where the interference of the operating system is silently suffered, the necessary number of observations is chosen offhand, and the results themselves are incorrectly treated as precise numbers [23][24]. For the group of complex benchmarks, we propose a technique for comparing results cluster-by-cluster, again in contrast with the current practice in middleware benchmarking, where the results are expressed as a simple value of throughput [9][18][20].

With the exception of clustering, the proposed techniques are fully automated. All the techniques are demonstrated on real-world examples of middleware performance evaluation. Additional details are available at http://nenya.ms.mff.cuni.cz and in [22].

6. Acknowledgements

This work is partially sponsored by the Grant Agency of the Czech Republic grant 102/03/0672.

7. References

[1] Beck K.: Extreme Programming Explained: Embrace Change, Addison Wesley, 1999.

[2] CAROL: Common Architecture for RMI ObjectWeb Layer, http://carol.objectweb.org.

[3] Chen E. J., Kelton W. D.: Simulation-based Estimation of Quantiles, Winter Simulation Conference 99, USA, 1999.

[4] The Community OpenORB Project, http://openorb.sourceforge.net.

[5] Distributed Systems Research Group: Open CORBA Benchmarking Project, http://nenya.ms.mff.cuni.cz/~bench.

[6] Distributed Systems Research Group: Vendor CORBA Benchmarking Project, http://nenya.ms.mff.cuni.cz/projects.phtml?p=cbench.

[7] DOC Group: TAO Performance Scoreboard, http://www.dre.vanderbilt.edu/stats/performance.shtml.

[8] DOC Group: The ACE Orb, http://www.dre.vanderbilt.edu/TAO.

[9] ECperf Specification, Version 1.1, Sun Microsystems, 2002, http://www.theserverside.com/ecperf.

[10] Enterprise JavaBeans Specification, Version 2.1, Sun Microsystems, 2003, http://java.sun.com/products/ejb/docs.html.

[11] Faber V.: Clustering and Continuous k-Means Algorithm, Los Alamos Science No. 22, 1994.

[12] Free High Performance ORB, http://omniorb.sourceforge.net.

[13] Gokhale A. S., Schmidt D. C.: Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks, IEEE Transactions on Computers Vol. 47 No. 4, 1998.

[14] Java Remote Method Invocation Specification, Sun Microsystems, http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.

[15] Krishna A. S., Balasubramanian J., Gokhale A., Schmidt D. C., Sevilla D., Thaker G.: Empirically Evaluating CORBA Component Model Implementations, OOPSLA 03 Middleware Benchmarking Workshop, USA, 2003, http://nenya.ms.mff.cuni.cz/projects/corba/oopsla-workshop.

[16] Leisch, F.: Bagged clustering, Adaptive Information Systems and Modeling in Economics and Management Science, 1999.

[17] Object Management Group, CORBA Specification 3.0.2, OMG formal/02-12-02, 2002.

[18] ObjectWeb Consortium: RUBiS: Rice University Bidding System, http://rubis.objectweb.org.

[19] Plášil F., Tůma P., Buble, A.: Charles University Response to the Benchmark RFI, OMG bench/98-10-04, 1998.

[20] Transaction Processing Performance Council: TPC Benchmark Web Commerce Specification 1.8, 2002, http://www.tpc.org.

[21] Tůma P., Buble A.: Open CORBA Benchmarking, SPECTS'01, USA, SCS, 2001.

[22] Bulej L., Kalibera T., Tůma P.: Regression Benchmarking with Simple Middleware Benchmarks, IPCCC'04 Workshop on Middleware Performance, USA, IEEE CS, 2004.

[23] Juric M. B., Rozman I., Hericko, M.: Performance Comparison of CORBA and RMI, Information and Software Technology Journal Vol. 42 No. 13, Elsevier Science, 2000.

[24] Boszormenyi L., Wickener A., Wolf H.: Performance Evaluation of Object Oriented Middleware - Development of a Benchmarking Toolkit, Euro-Par'99, France, LNCS 1685, Springer Verlag, 1999.