CORBA Benchmarking: A Course with Hidden Obstacles

Adam Buble, Lubomír Bulej, Petr Tůma
Charles University, Faculty of Mathematics and Physics
Department of Software Engineering
Malostranské nám. 25, Prague, Czech Republic
Phone: +420 221 914 267, fax: +420 221 914 323
{adam.buble,lubomir.bulej,petr.tuma}@mff.cuni.cz

Abstract

Numerous projects have evaluated the performance of CORBA middleware over the past decade. Interestingly, many of the published results are either gathered or analyzed imprecisely. We point out common causes of such imprecision related to the gathering of timing information and the effects of warm-up, randomization, cross talk and delayed or hidden functionality, and demonstrate their impact on the results of the evaluation. We also present suggestions related to reporting the results in a manner that is more relevant to the evaluation.

1. Introduction

In 1991, CORBA [1] emerged as middleware architecture for object communication in heterogeneous distributed environments. As the architecture developed and established itself as an industrial standard, a need for assessing the performance of its implementations became apparent. In response to this need, a number of benchmarking projects appeared [2][3]. Although different in many aspects, these projects had to solve a number of common problems, which prompted attempts to define a common benchmarking framework [4][5][6].

Unfortunately, this framework never materialized. Thus, many benchmarking projects still keep reinventing the technical details of the benchmarks, often deceptively mundane in appearance, and end up committing the same mistakes.

We point out some of the common mistakes committed in benchmarking projects and discuss their impact on the results of the performance evaluation. Section 2 discusses the causes of imprecisely gathered or analyzed results in dedicated subsections. Section 3 follows with comments on reporting the results. Section 4 concludes the paper.

2. Causes of imprecision

2.1. Imprecise timing information

The absolute majority of benchmarks involve measuring time. It is therefore imperative that a precise source of timing information is used. Benchmarking CORBA middleware typically involves measuring actions that are as short as tens of microseconds. Unless we accept the loss of information incurred by benchmarking a sequence of actions and collecting averaged timing information, this disqualifies sources with millisecond or worse resolution.

Use of such an inappropriate source of timing information can be found in [13], where the authors describe measurements carried out on Intel Pentium III, 660MHz, 100Mbit LAN, Windows NT, JacORB. In their test application, the authors acquire timing information in various stages of method invocation, both on the client and the server, using the `System.currentTimeMillis()` method from Java SDK. The actual resolution of this source on the given platform was 10 ms, but the duration of a remote method invocation on the same platform is significantly shorter than 10 ms, probably under 1 ms. Using this time source can therefore lead to incorrect results.

The use of time sources with a resolution of 1ms is actually rather common, probably because of their wide availability. It is important to note that such timing information is usually collected for operations that are repeated to obtain an aggregated average. Such an average, however, may mask the actual behavior of the middleware, as discussed in section 3.1.

A related difficulty stems from the distributed nature inherent to CORBA. Timing information collected on different nodes typically cannot be related unless some means of time source synchronization among the network nodes are employed.

0-7695-1926-1/03/$17.00 (C) 2003 IEEE

In [13], the authors performed a measurement with the benchmark running on several nodes and subtracted timestamps collected on different nodes from each other. When some of the results came out negative, the authors attributed this anomaly to network latency being lower than the resolution of the time source.

In our experience, a good source of timing information on the Intel platforms is the timestamp counter accessible via the `RDTSC` instruction. On Solaris platforms, timing information can be acquired using the `gethrtime()` system call. On Windows platforms, timing information can be acquired using the `QueryPerformanceCounter()` system call. To our knowledge, there are no generally available means of time source synchronization among the network nodes that would not require specialized hardware such as GPS.

2.2. Insufficient warm-up

To obtain stable results, it is typically necessary to let the system perform the action to be benchmarked for some time before the timing information is collected. This allows minimizing or eliminating the influence of various factors that can render the results incorrect. Typically, these factors include system activity related to application startup, priming of caches, stabilization of adaptive resource allocation algorithms in response to system load or network traffic, just in time compilation.

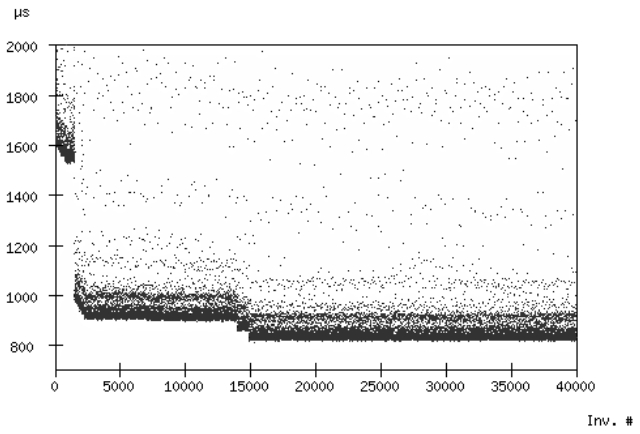

Figure 1. Dependency of roundtrip time on the length of warm-up in number of invocations

To document the necessity of warm-up in benchmarking applications, we show results of a simple test consisting of repeated invocations of a `Pong()` method on a single `Ping` object. The platform used in the test was Intel Pentium III, 1GHz, 512MB RAM, Windows 2000, Sun JVM 1.3.1 and Orbix 2000 5.1 for Java. Figure 1 plots the roundtrip times of 40000 invocations, with the effect of just in time compilation performed by the JVM both on the client and the server clearly visible. We can conclude that in this particular setup, a warm-up of at least 15000 invocations is necessary.

An example of insufficient warm-up can be found in [10][11], where the authors perform the measurements without warm-up and for 200 iterations only.

2.3. Insufficient randomization

The benchmarks that analyze the impact of a specific factor on the overall performance can be influenced by implicit choices related to this factor. As an example, consider a benchmark that evaluates dependency of the roundtrip time of an invocation on the number of active objects or concurrent threads. The implementation of such a benchmark needs to choose the object or the thread to perform the measurement on, and this choice can influence the observed results.

The problem of implicit choice cannot be eliminated simply by collecting data for all possible choices, since the amount of data would be too large and the collection itself could distort the measurement. From our experience, it is more practical to make such choices random and repeat the measurement several times to make sure the collected data will not be distorted by a systematic choice.

2.4. Interfering cross talk

Rather than devising a benchmark that evaluates the system as a whole, with CORBA middleware it is usually preferable to construct a collection of benchmarks that measure the dependency of performance on isolated factors [7]. It may be difficult, however, to isolate the influencing factors entirely.

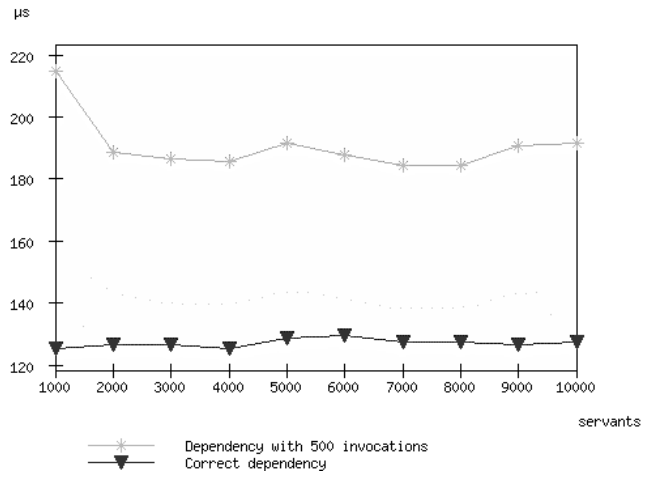

Figure 2. Dependency of roundtrip time on the number of active objects

As an example, consider a benchmark that measures the dependency of the roundtrip time of an invocation on the number of active objects. The graph in Figure 2 displays two

versions of such a dependency for a platform with Dual Intel Xeon, 2.2GHz, 512MB RAM, Linux 2.4 and ORBacus 4.1. The decreasing dependency was measured by a benchmark that collected the timing information from 500 random invocations in each step of 1000 newly created objects. The constant dependency was measured by a benchmark that issued an invocation to each of the newly created objects before collecting the timing information.

The difference between the two versions is caused by the fact that the roundtrip time of an invocation also depends on whether the invocation happened to be the first for the invoked object, as can be seen in Figure 3. The cross talk of these two factors thus distorts the results to a degree that depends on the percentage of invocations that happened to be the first for the invoked object.

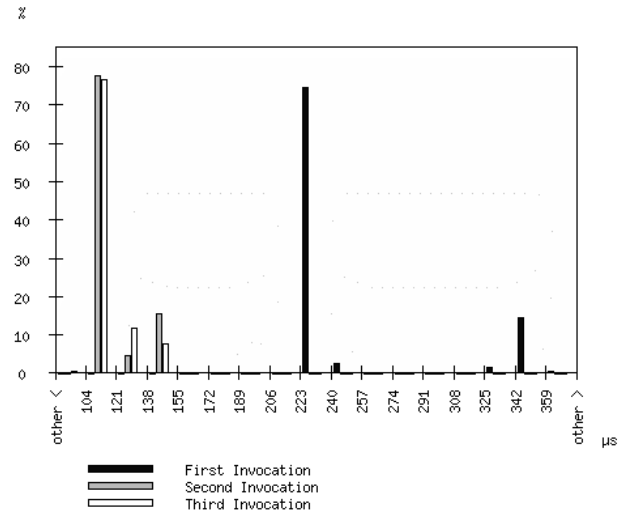

Figure 3. Differences between the first and subsequent method invocations

2.5. Delayed functionality

Certain actions performed by the CORBA middleware may not actually happen when issued. Instead, they can be postponed until their result is used. A typical example of this is unmarshalling of complex data types that provide overloadable access operators or methods. Figure 4 shows the difference in roundtrip times when invoking two methods that receive the same instance of a 1024-octets-long array encapsulated in `CORBA::Any` as their input argument. One method accesses this instance, while the other method accesses an identical instance that was not passed as the input argument. The platform used was Dual Intel Xeon, 2.2GHz, 512MB RAM, Linux 2.4 and omniORB 4.0.

A situation where this particular problem needs to be considered can be found in [12]. Here, the authors present their performance framework, which uses `CORBA::Any` along with other IDL types. The servant implementation, however, does not access its arguments, in effect not measuring part of the unmarshalling mechanism of the middleware.

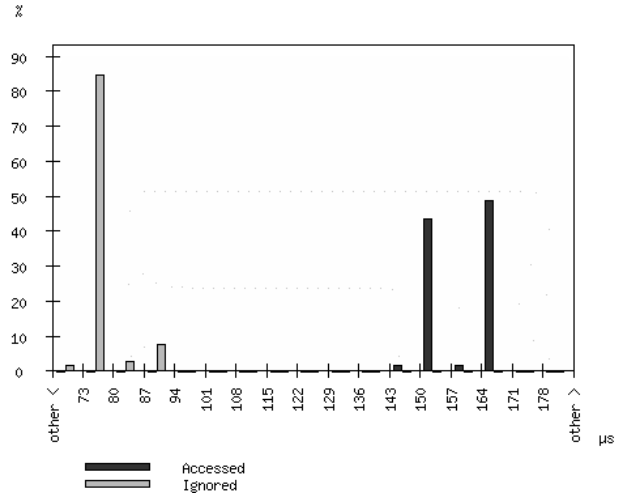

Figure 4. Differences in accessing CORBA::Any argument

2.6. Hidden functionality

Even carefully designed benchmarks can face a situation where the actual measurement interferes with some hidden mechanism of the benchmarked middleware. An example documenting such interference is marshalling and unmarshalling of `char` and `string` types, which may include character set conversion. Brokers that implement the character set conversion will be handicapped in benchmarks that use `char` and `string` types. An example of a difference between MICO 2.3.1, which does not implement character set conversion, and ORBacus 4.0, which does, on Intel Celeron, 466MHz, 128MB RAM, Windows NT, can be seen on Figure 5.

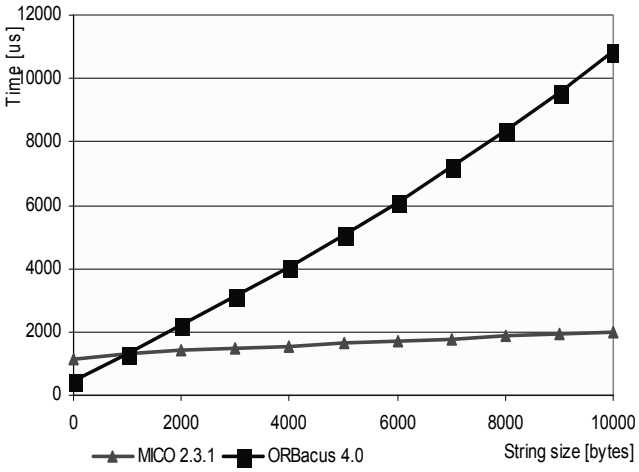

Figure 5. Dependency on support of character set conversion

A very similar situation may arise when comparing CORBA middleware with different middleware technologies such as RMI. In [10], the authors compare RMI with CORBA, besides other also on `string` types. Incidentally, the JavaIDL CORBA middleware implementation that the authors used does not support character set conversion, and thus the particular results are comparable.

3. Reporting the results

The issue of reporting results was debated in depth in [6][7] and other papers. Here, we would like to emphasize only two particular details that we believe are helpful for correct interpretation of the results.

3.1. Reporting averaged results

In many cases, results reported as averages do not provide all the information needed for their correct interpretation. Often, reporting averages simplifies the collecting and processing of results, but we find that it is very useful to also report results as box-and-whisker plots and statistical distribution plots.

For an example, see Figure 6, which shows the dependency of the invocation time on the number of concurrent threads on Dual Intel Xeon, 2.2GHz, 512MB RAM, Linux 2.4 and omniORB 4.0. Compare this with the same results distorted by averaging over 100 invocations in Figure 7. Averaging over 100 invocations is used for example in [12].

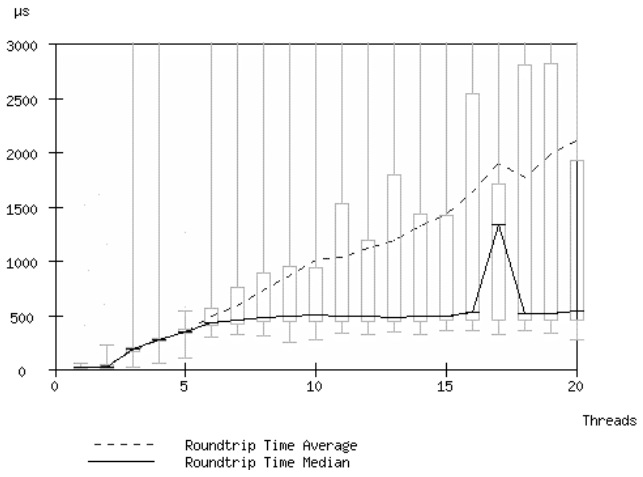

Figure 6. Dependency of roundtrip time on the number of threads

3.2. Reporting multithreading results

Typically, the goal of benchmarks that employ multiple threads is the assessment of the achievable degree of parallelism and the behavior of the middleware under high load. This is done by measuring the dependency of the roundtrip time of an invocation on the number of threads that issue the invocation concurrently.

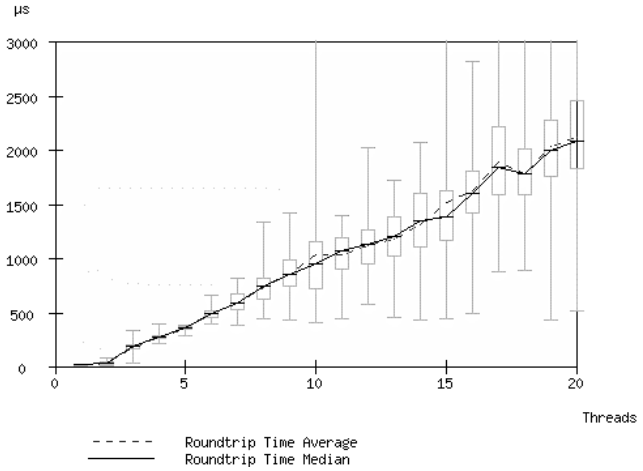

Figure 7. Dependency of roundtrip time on the number of threads, averaged results

When measuring invocations that allow a high degree of parallelism, the overriding factor that influences the results is the processor power. During the measurement, a single thread will execute actions until it uses up the quantum allocated to it by the operating system. The thread will then be preempted by another thread and only regain control after all other threads use up their quanta. In this situation, the observed average of the invocation time will increase linearly with the number of threads. The observed median of the invocation time will remain constant when the invocation is relatively short compared to the quantum, and will increase linearly otherwise.

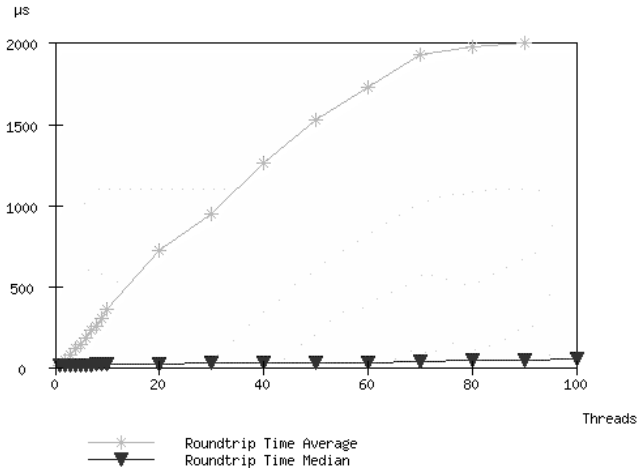

Figure 8. Dependency of roundtrip time on the number of client threads

An example of this behavior for repeated invocations of a `Pong()` method on a single `Ping` object on Intel Xeon,

2.2GHz, 512MB RAM, Linux 2.4 and omniORB 4.0, is on Figure 8.

When measuring invocations that only block to a relatively small degree, the behavior will not differ significantly from the previous case. In addition to the linear dependency described above, an increase in the median and average values will be observed. This increase will account for the time spent by waiting, in effect characterizing the achievable degree of parallelism.

When measuring invocations that block to such a degree that the sum of the time spent waiting in all threads nears or exceeds the duration of a single invocation, the behavior of the system will change considerably from the previous two cases. The threads employed by the benchmark will form a scheduling convoy. As in the other two cases, an increase in the median and average values will still be observed. Unlike in the other two cases, however, this increase will not account for the time spent by waiting, but for the overhead incurred by the queuing mechanisms that accommodate the waiting threads and their resources.

An example of this behavior for an invocation that creates a servant is on Figure 9. The platform used was Sun Ultra 60, Dual SPARC, 360MHz, 512MB RAM, Solaris 7 and Orbix 2000 5.1.

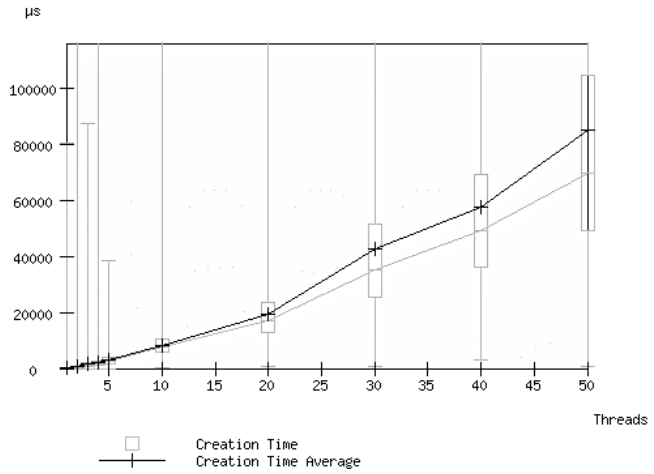

Figure 9. Dependency of servant creation time on the number of client threads

Obviously, rather than plotting the average invocation time observed by individual threads, as is done in [14], it might be more useful to plot the ratio of the average invocation time to the number of threads. Such a plot helps suppress the linear dependency that only expresses the obvious fact that the processor power is distributed among the individual threads, and emphasize the increases in average invocation time that characterize the middleware implementation.

An example of such a plot for an invocation that creates and deletes a servant can be found below on Figure 10. The platform used was Sun Ultra 60, Dual SPARC, 360MHz, 512MB RAM, Solaris 7 and Orbix 2000 5.1.

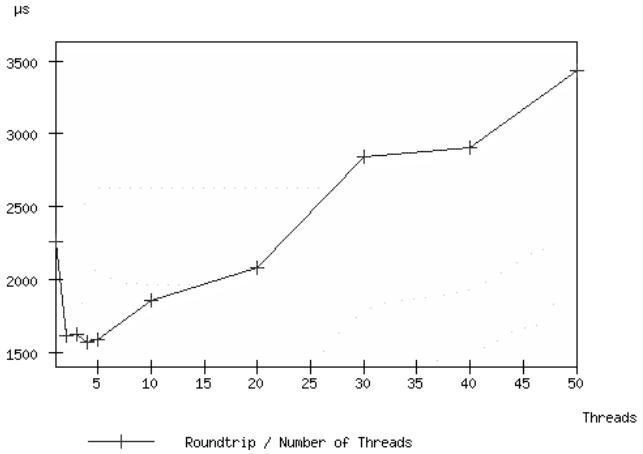

Figure 10. Ratio of average roundtrip time to the number of threads

4. Conclusion

We have analyzed several aspects of CORBA middleware benchmarking and identified issues that can distort or even invalidate the measurement results. Rather than attempting to make the list of issues complete or exhaustive, we focus on those points that are important based both on our experience and on the evidence of other published CORBA benchmarking projects.

In addition to pointing out the selected issues, we also suggest how to avoid common mistakes related to those issues. The suggestions for benchmarking in general concern the acquisition of timing information, the necessity of warm-up phase in benchmarking applications and the randomization of invocation patterns. The suggestions for benchmarking CORBA middleware concern cross talk between different influencing factors and the existence of delayed or hidden functionality that can distort the measurement. Finally, the suggestions for reporting results concern reporting statistically more relevant data and interpretation of data collected from multithreaded benchmarks.

References

[1] Object Management Group, *The Common Object Request Broker: Core Specification*, version 3.0.2, OMG formal/02-12-02, Dec 2002

[2] Distributed Systems Research Group, *CORBA Comparison Project*, Project Extension Final Report, Charles University, Prague, 1999

[3] Callison, H., R., Butler, D., G., *Real-time CORBA Trade Study*, Boeing, 2000

[4] Plasil, F., Tuma, P., Buble, A., *Charles University Response to the Benchmark RFI*, OMG bench/98-10-04, Oct 1998

[5] University of Helsinki, *Response to ORBOS PTF Benchmark RFI*, OMG bench/98-10-03, 1998

[6] Object Management Group, *White Paper on Benchmarking*, OMG bench/99-12-01, 1999

[7] Tuma, P., Buble, A., *Open CORBA Benchmarking*, Proceedings of the 2001 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '01), published by SCS, Orlando, FL, USA, Jul 2001

[8] 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, Apr 1998

[9] Juric, M., B.; Rozman, I., *Java 2 RMI and IDL Comparison*, Java Report, No. 2, SIGS Publications, Feb 2000

[10] Juric, M., B.; Rozman, I., *RMI, RMI-IIOP and IDL Performance Comparison*, Java Report, Vol. 6, No. 4, SIGS/101 Publications, Apr 2001

[11] Juric, M., B., Rozman, I., Stevens, A., P., Hericko, M., Nash, S., *Java 2 Distributed Object Models Performance Analysis, Comparison and Optimization*, Proceedings of ICPDAS '00, Iwate, Japan, IEEE Computer Society Press, pg. 239-246, Jul 2000

[12] Juric, M., B., Welzer, T., Rozman, I., Hericko, M., Brumen, B., Domajnko, T., Zivkovic, A., *Performance Assessment Framework for Distributed Object Architectures*, ADBIS '99, LNCS 1691, Springer Verlag, Sep 1999

[13] Grahn, H.; Holgersson, M., *An approach for performance measurements in Distributed CORBA applications*, Applied Informatics (AI 2002), Innsbruck, Austria, 2002

[14] Boszormenyi, L., Wickener, A., Wolf, H., *Performance Evaluation of Object Oriented Middleware - Development of a Benchmarking* Toolkit Euro-Par'99, 5th International Euro-Par Conference, Toulouse, France, LNCS 1685, Springer Verlag, 1999