

# Overview of the CORBA Performance

Petr Tůma, Adam Buble  
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  
e-mails: petr.tuma@mff.cuni.cz, adam.buble@mff.cuni.cz

**Abstract:** CORBA has been established as one of the most common middleware today. Its language transparency and strong industrial background makes it promising for the future. However, choosing the right implementation is not easy. The implementations vary in features and performance and user should carefully choose the one to use. We present a benchmarking suite, which is simple and yields results that are easy to understand. The results can be combined to assess the performance of more complicated application. Finally, we present an overview of performance of today's C++ CORBA brokers.

**Keywords:** CORBA, benchmarking, performance.

## 1 INTRODUCTION

In 1991, CORBA emerged as a promising middleware architecture for object communication in potentially heterogeneous and distributed environments. As the architecture developed and established itself as an industrial standard, a number of various implementations appear. As implementations vary in features and performance, it became hard to choose the right one for the user. The features are usually evaluated easily by inspecting the vendor's web site, but the performance is hidden from user. To resolve this issue number of benchmarking projects appeared [1][2][3][4]. These differed in a number of aspects, including the range of the benchmarked functions, the setup of the benchmarking environment, and the presentation of the benchmarked results. This made the interpretation of the benchmarking results difficult, especially when it was necessary to relate results of different benchmarks. The problem was identified by OMG in the 1998 Benchmarking PSIG Request For Information [5], and received a number of responses [6][7][8][9] that were summarized in the OMG White Paper On Benchmarking [10].

The White Paper styles itself as "a first step towards a uniform benchmarking methodology to be applied for various OMG technologies", and focuses mostly at outlining the issues related to benchmarking and planning the future work. This was expected to take a shape of a benchmarking framework, a benchmarking methodology, a repository for public benchmark algorithms, and a tool for automated benchmark execution. Although none of the plans materialized so far, the White Paper still voices a conviction of its contributors that there is a need for benchmarks that would be open, well understood, and easy to measure.

In typical benchmarking projects, these attributes are often considered of secondary importance. This is because the projects are done in response to particular needs of the involved parties, and those needs rarely include openness. Thus, the projects might stop short of publishing complete results [4][11][12][13], complete methodology [3], or complete sources [14]. Even if published, the project results are generally difficult to interpret, and impossible to relate to each other.

Our experience gathered in projects with MLC Systeme (now Deutsche Post Com) [4][11][12][13], Bull Soft (now Evidian) [11], IONA Technologies, and others, induces in the paper [15]. In this paper we simplify the proposed benchmarking suite to target the broker users. The suite is implemented and available at our web pages.

In chapter 2, we present an open benchmark suite for evaluating performance of CORBA middleware. In chapter 3, we follow with the guidelines on executing the suite and reporting the results.

## 2 SIMPLE OPEN BENCHMARK SUITE

The obvious goal of an open benchmark suite is to cover all commonly used broker functionality. This goal, however, has the potential of yielding too large a suite, which goes against the requirements of well understood and easy to measure benchmarks that was expressed in the call for an open benchmark suite [10]. To tackle this problem, we have decided to consider two different target audiences for the benchmark results, broker vendors and broker users:

**Broker vendors** represent an audience that is interested in the benchmark results mainly because they want to assess the performance of their particular broker, for example to identify a performance related bug or a bottleneck, or to evaluate a performance impact of a modification done during broker development.

To be able to satisfy these requirements, a benchmark suite needs to cover every possible aspect of broker performance in detail, including aspects specific to a particular broker implementation. The benchmark results can be very technical, because the people reading it are likely to be familiar with the technical details of the broker.

**Broker users** represent an audience that is interested in the benchmark results mainly because they want to understand what performance can be expected from a broker in their particular application, for example to decide what broker implementation to use, or to architect an application based on the available broker performance.

To be able to satisfy these requirements, a benchmark suite needs to be easily understandable, because the people reading it will not necessarily be experts in broker architecture and performance. The benchmark results can be simplified through generalization, although it is important to clearly note the cases where the generalization might not work.

Although the two audiences can occasionally benefit from the benchmark results targeted at the other audience, the principal difference is significant enough to warrant separate benchmark suites.

The benchmark suite for the broker vendor audience outlined in [15][16] provides a detailed overview of most factors influencing the broker performance. In its evolving form, it was successfully used in several benchmarking projects [4][13], and formed a basis for an EJB benchmark suite [11][12].

The benchmark suite for the broker user audience outlined in following sections provides a rough overview of the basic factors influencing the broker performance. When implemented for C++ brokers, the suite proved easily portable, typically requiring a change of under 10 lines of code per broker. The implementation is available at <http://nenya.ms.mff.cuni.cz/~bench>.

## 2.1 Performance Factors

The chief purpose of a CORBA broker implementation is mediating communication between clients and servers. The performance of the broker can therefore be expressed in a straightforward manner using metrics such as invocation time or communication throughput. To analyze the individual performance factors that influence the overall result of these metrics, we need to follow the invocation path of a typical broker.

Typical invocation path consists of

- invocation,
- marshalling,
- network transport,
- receiving,
- servant locating,
- unmarshalling,
- processing.

It is obvious, that each of these phases depends on various factors and efficiency of implementation, and can be therefore evaluated. The return phase of the invocation is similar as far as the performance factors influencing the overall result go.

### 2.1.1 Client Side

The invocation on the client side can use one of three available invocation mechanisms, namely the normal static invocation, the asynchronous invocation as specified in CORBA Messaging, and the dynamic invocation using the Dynamic Invocation Interface. All three mechanisms proceed by marshaling the arguments into a request.

### 2.1.2 Network Transport

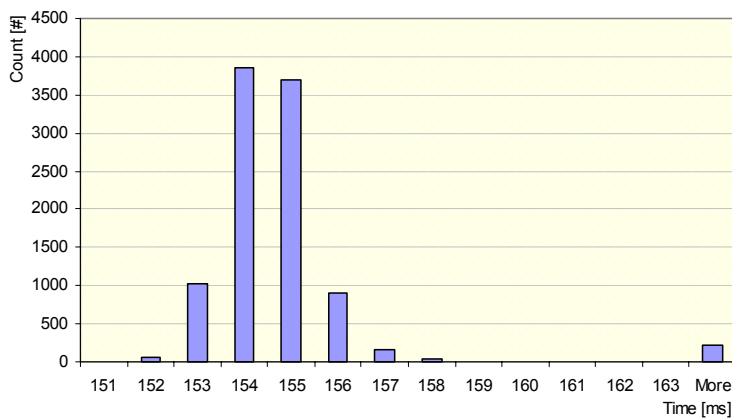
Besides transporting the invocation data in a layout that is typically compliant with the IIOP standard [17], and therefore varies little from broker to broker, this phase of the invocation can also include setting up or looking up the connection and processing the forwarding messages. These points do not lend themselves well to benchmarking, as they do not occur in continuous load scenarios, and it is difficult to isolate their effect on the observed performance.

### 2.1.3 Server Side

The invocation on the server side begins by receiving the request from network layer. The request is dequeued, can be stored in some buffer and then dispatched to the proper object adapter and from there to the proper servant. After that, two invocation mechanisms are available, namely the synchronous static invocation and the synchronous dynamic invocation using the Dynamic Skeleton Interface. Both mechanisms proceed by unmarshaling<sup>1</sup> the arguments from the request.

## 2.2 Invocation Benchmarks

Basic invocation benchmarks measure the distribution of delivery and roundtrip times for a simple method invocation with no arguments in all available invocation modes of the broker. The results can be given in a form of a graph depicting the statistical distribution of the delivery and roundtrip times. However, for a specific broker implementation, the delivery and roundtrip times are typically very close to each other both in absolute values and in statistical distribution regardless of the invocation mode used, with the dynamic invocation models being slightly slower than the static invocation models. It is therefore an acceptable simplification to present the results of a benchmark that uses a single invocation model only (Figure 1).



**Figure 1** Distribution, roundtrip time, static invocation, **TAO 1.1.13**, Linux 2.4.2, Dual Intel Pentium III 800 MHz, 512 MB RAM

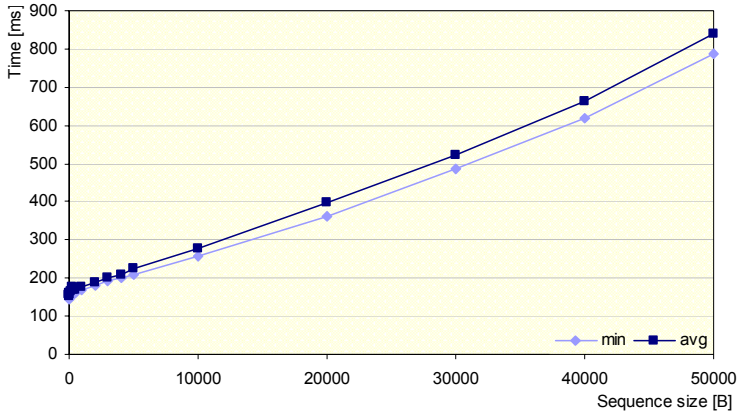
Important result gained from the benchmark is median of invocation time and the distribution of it. The median is more valuable than the average invocation time, as it shows the most probable invocation response time. However it is hard to measure in some benchmarks.

For brokers that support Quality of Service settings through policies defined in CORBA Messaging, additional benchmarks have to be developed to test the individual policies. Because the CORBA Messaging is fairly new, and many current broker implementations do not support it, we refer the reader to specific Quality of Service research [18], and do not attempt to simplify the benchmarks for this particular case.

## 2.3 Marshalling Benchmarks

Marshalling benchmarks should measure the dependence of the delivery and roundtrip times for a simple method invocation on the type, encapsulation, size and direction of the data passed as arguments of the invocation. For a specific broker implementation, the delivery and roundtrip times typically do not vary significantly with the type, encapsulation and direction of the data. It is therefore an acceptable simplification to omit the results of benchmarks that measure the impact of these factors. A notable exception is passing character data in an environment that supports code page conversion, which can introduce a significant overhead. Because of its special nature, we believe the benchmark measuring the overhead for code page conversion should be a part of the benchmarks for broker vendor audience. Hence we have dramatically reduced the size of the suite to consist only of single major benchmark – dependency of the invocation roundtrip time on the size of the data.

<sup>1</sup> Unmarshalling is not the domain of server side only – client unmarshalls the replies from server. We list unmarshalling points under the server side because of clarity.



**Figure 2** Roundtrip time for growing sequence<octet>, direction in, static invocation, **MICO 2.3.6**, Linux 2.4.13, Dual Intel Pentium III 800 MHz, 512 MB RAM

The delivery and roundtrip times typically depend linearly on the size of the data. It is therefore an acceptable simplification to present the results of a benchmark that uses a single type, encapsulation and direction of the data only (Figure 2). To avoid conversion slow down, we have selected to transfer IDL octet type. To easily adjust the size of the transferred data, we have choose the IDL sequence type to transmit.

The results are given in a form of a graph depicting the dependency of the roundtrip times on the size of the data (Figure 2). The graph depicts either average and minimum roundtrip time for the given sequence length/size. It is good for checking the correct behavior of the implementation.

#### 2.4 Receiving Benchmarks

These benchmarks are closely tied to network setting and processing benchmark (multithreading). Please see Parallelism Benchmarks in section 2.7.

#### 2.5 Dispatcher Benchmarks

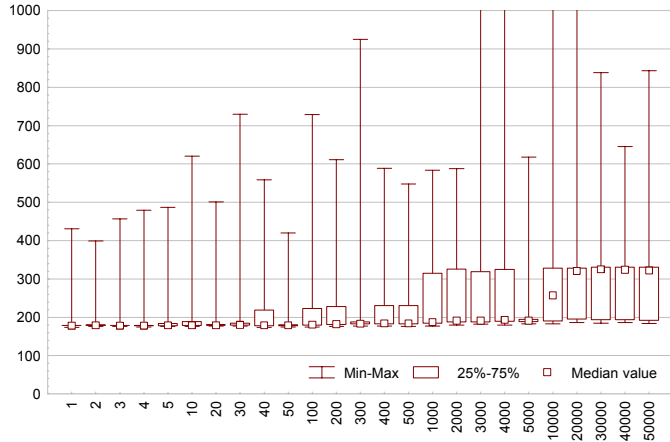
Dispatcher benchmarks should measure the dependence of the invocation times for a simple method invocation on the complexity of the interface the invoked method belongs to, the complexity of the object adapter hierarchy the invoked instance belongs to, and the servant registration policy and the number of servants registered by the object adapter of the invoked instance. This would be too many benchmarks, so we have made several simplifications to conduct only those benchmarks which yield important results.

Of the complexity of the interface, the factors that can influence the invocation times are the number of methods in the interface, because the method gets looked up at the server during each invocation, and the length of the method name, because the name is passed and looked up during each invocation. These do not affect the performance heavily, as the interfaces are usually not too large or complicated.

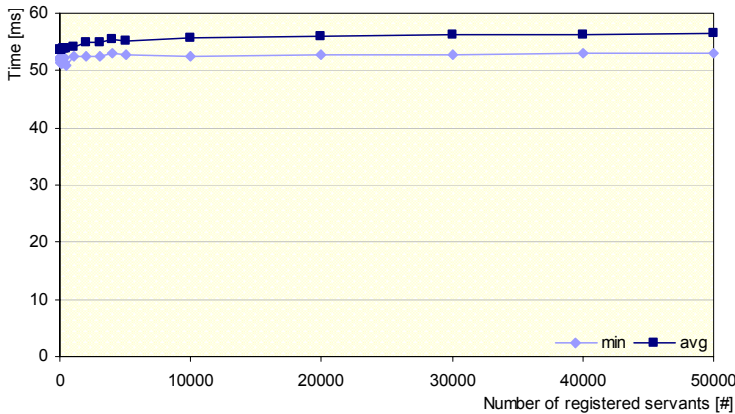
Of the complexity of the object adapter hierarchy, the factors that can influence the invocation times are its width and depth. As far as our measurements show, there is nearly no dependency on these factors.

Of the servant registration policies, the most interesting one from the performance point of view is the RETAIN policy, which instruct the POA to retain a map of active servants that is automatically searched during each invocation. This leads to a straightforward benchmark, whose results are given in a form of a graph depicting the dependence of the invocation times on the number of servants registered in the map (Figure 3). It is therefore an acceptable simplification to present the results of a benchmark that measures the influence of the number of registered servants only (Figure 4).

As a side note, many broker implementations use algorithms such as imperfect hashing, sequential search, or sequential compare, to perform some of the operations described in this section. These algorithms can all exhibit the best case or worst-case behavior that is significantly different from an ordinary case. The benchmarks should take care to measure and report both the ordinary and the extraordinary behavior, possibly using the box and whisker graphs (Figure 3) rather than usual line graphs (Figure 4). However common data processing tools (such as MS Excel) does not have the ability to create box and whisker graphs. Such a graph is much better as it does not show the average time (which is not stable in statistical sense), but median, quartiles, minimum and maximum.



**Figure 3** Roundtrip times, void ping() for increasing number of servants, static invocation, **ORBacus 4.0.4**, Linux 2.2.16, Dual Pentium III 800 MHz, 512 MB RAM



**Figure 4** Roundtrip times, void ping() for increasing number of servants, static invocation, **omniORB 3.0**, Linux 2.4.2, Dual Intel Pentium III 800 MHz, 512 MB RAM

The average invocation time (and median as well) for the benchmark usually does not increase steeply. The reasoning is that the active object map within the POA should be implemented using hash-table, which ensures the access to the servant in constant time. The steep average invocation time together with usually non-steep minimum time clearly shows inappropriate implementation of active object map (see Figure 5).

## 2.6 Processing Benchmarks

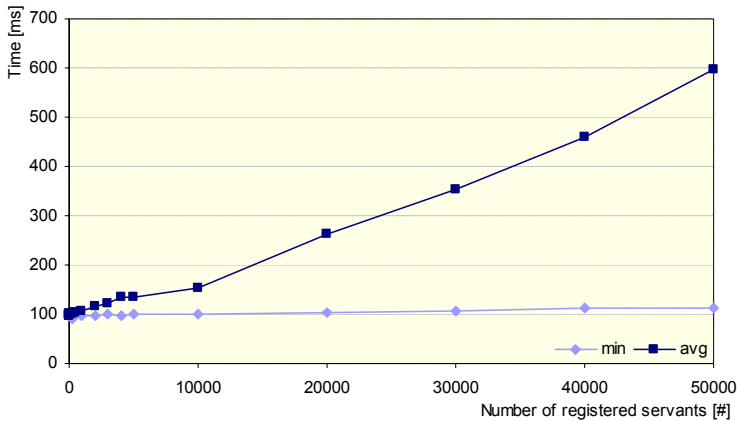
Processing benchmarks should measure impact of individual threading strategies provided by CORBA middleware. The ORBs usually implements various threading strategies, some of them are only single threaded. It is impossible to define common set of threading models and it is therefore impossible to compare individual ORBs mutually. However, testing multi-threading is crucial for users. We suggest therefore to test default threading strategy, as the ORB vendor usually selects the most common strategy as the default one. Moreover, we do not forbid the user to change the threading strategy before starting benchmarking. The suite should detect the actual threading model and report it in the result.

Processing benchmarks are tied to client setting – either many clients, or many threads within single client connect to a single server. Because of the inherent problem of separating both sides from each other and mainly because of simplicity (user does not usually have hundreds of computers), we suggest simplified but still valuable benchmark in section 2.7.

## 2.7 Parallelism Benchmarks

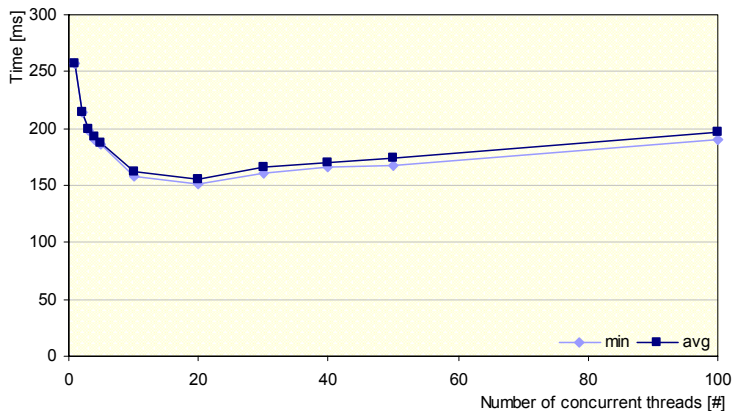
Parallelism benchmarks measure the dependence of the invocation times for a simple method invocation on the number of threads and number of clients that issue or handle the invocations in parallel. Besides the invocation times as perceived by the

individual threads, the benchmarks also measure the invocation rate as perceived by the system. The invocation times perceived by a single thread grow whenever the thread is blocked waiting for another thread, and thus describe the degree of parallelism achieved during invocation (Figure 6). The invocation rate perceived by the system increases as the ability of the system for parallel execution gets increasingly exploited, up to the point where it is fully exploited, from which the invocation rate slowly decreases as the overhead associated with the parallel execution increases. The results are given in a form of a graph depicting the dependence of the invocation times and the invocation rate on the number of threads and number of clients that issue or handle the invocations in parallel.



**Figure 5** Roundtrip times, void ping() for increasing number of servants, static invocation, **VisiBroker 4.5**, Windows NT 4.0 SP 6, Dual Intel Pentium III 800 MHz, 512 MB RAM

The simplification stems from the fact that a benchmark where a large number of clients issue invocations is difficult to execute. In a local setup, the load generated by the clients interferes with the server. In a network setup, the network bottlenecks can do the same. Also, the cost of maintaining a benchmarking environment capable of executing a large number of clients in parallel is not trivial. To keep the benchmark suite easy to execute, we can only include the benchmarks that use multiple threads rather than multiple clients.



**Figure 6** Roundtrip time, void ping() for increasing number of threads, static invocation, **Orbix 2000 1.2.1**, Linux 2.4.13, Dual Intel Pentium III 800 MHz, 512 MB RAM

Depending on the specific broker implementation, the parallelism benchmarks need to be run several times or extended to cover all threading strategies. Also, the distribution of the system capacity among the individual threads needs to be evaluated [19].

## 2.8 Miscellaneous Benchmarks

A broker also provides functions that are not directly related to invocation. Typical examples of these are insertion and extraction of data into and from anys, string allocation, or dynamic any inspection. Straightforward benchmarks of these functions can be incorporated into the benchmarks for the broker vendor audience, but are not significant enough to be included in the benchmarks for the broker user audience.

## 3 EXECUTING AND REPORTING

The paramount criteria for executing the benchmarks and reporting the results are relevancy and repeatability. A benchmark needs to be relevant in that the measurements must reflect typical modes of operation in the selected problem domain [10], and repeatable in that the target audience must be able to reproduce the results if necessary [20].

The results of the benchmarks described in section 2 are not immediately relevant in the sense considered here, because the mode of operation used corresponds to none but most trivial problem domains. The chief differences are testing individual performance factors separately, testing with no background load, and testing with ideal network conditions.

### 3.1 Combining Performance Factors

A typical approach to providing relevant results is adjusting the benchmark so that it executes a mixture of operations similar to that executed by existing applications [21]. This is difficult to do for a broker, because the typical modes of operation, and therefore also the importance of the performance factors, vary significantly [10]. Our approach therefore is to characterize a specific mode of operation in terms of the individual performance factors, and then combine the results of the individual benchmarks based on this characterization.

The performance factors considered by the benchmark suite are the simple invocation time, the size of the invocation arguments, the number of the registered servants, and the number of the active threads.

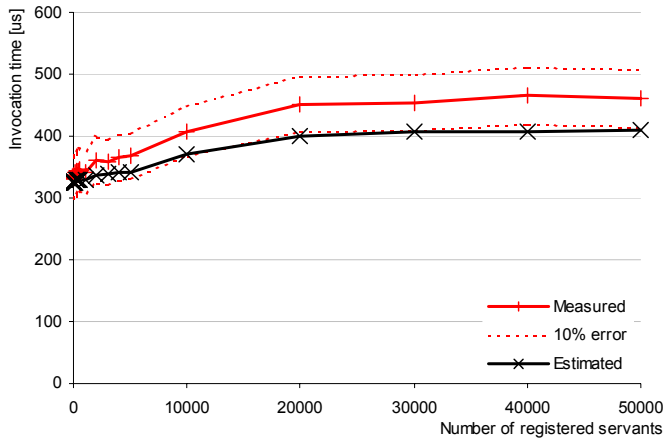
The simple invocation time will be a part of each result. The size of the invocation arguments adds an overhead that represents time spent marshalling and transporting the arguments. This overhead is additive in the sense that adding the arguments to an invocation means adding the overhead to the invocation time. The number of the registered servants adds an overhead that represents time spent marshalling and transporting longer object references and looking up in larger tables. This overhead is also additive. The number of the active threads adds a scaling factor that represents the degree of parallelism of invocations.

Knowing the typical size of invocation arguments *size*, number of registered servants *servants*, and number of active threads *threads* for a specific mode of operation, the results of the individual benchmarks can therefore be combined to form an estimated relevant result using a simple formula:

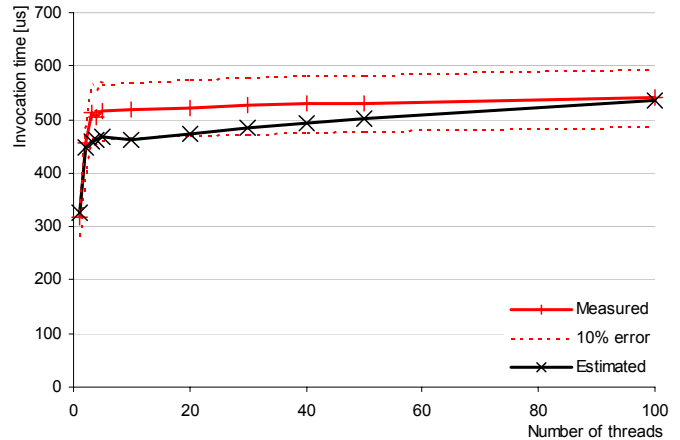
$$\begin{aligned} \text{estimated\_time} = & \\ & (\text{bench\_simple} + \\ & \quad \text{bench\_size}(\text{size}) - \text{bench\_size}(0) + \\ & \quad \text{bench\_servants}(\text{servants}) - \text{bench\_servants}(1)) \times \\ & \text{bench\_threads}(\text{threads}) / \text{bench\_threads}(1) \end{aligned}$$

Where *bench\_simple* denotes the simple invocation time (section 2.2), *bench\_size(x)* denotes the invocation time for arguments of size *x* (section 2.3), *bench\_servants(x)* denotes the invocation time for *x* servants (section 2.4), and *bench\_threads(x)* denotes the invocation time for *x* threads (section 2.7).

The estimate is based on the assumption that the factors are largely orthogonal. Our measurements indicate this condition holds until the factors start influencing each other through exhaustion of shared resources. The precision of the estimated relevant result as opposed to the measured one is illustrated at Figure 7 and Figure 8.

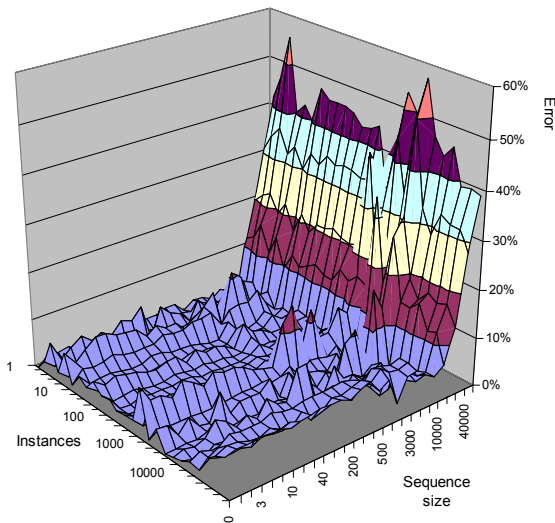


**Figure 7** Roundtrip times, void ping() for increasing number of servants, static invocation, **ORBacus 4.0.4**, Linux 2.2.16, Dual Pentium III 800 MHz, 512 MB RAM



**Figure 8** Roundtrip times, void ping() for increasing number of threads, static invocation, **ORBacus 4.0.4**, Linux 2.2.16, Dual Pentium III 800 MHz, 512 MB RAM

Special benchmark used just to test the presented approach acknowledges its correctness. The benchmark is a complete version of invocation-marshalling-dispatching-threading benchmark mixture. All the possible combinations (within specified ranges) are tested and reported. These measured results are then compared to the estimated values. The results are encouraging as the average error is usually less than 10 % (except of VisiBroker 4.5 where is usually around 20 %). As stated before, the estimation works fine only when there are no exhausted/overloaded resources (mostly CPU). For an average error see Figure 9. The part of the graph where huge sequences are handled is loaded, because the marshalling makes significant time on the CPU. The estimate could be normalized (see section 3.2), however it is hard to obtain specific coefficients.

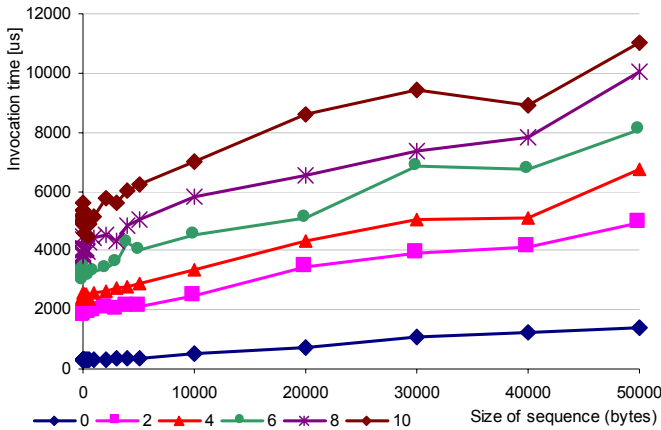


**Figure 9** Absolute error for estimating, 10 threads, **ORBacus 4.1.0**, Linux 2.4.17, Dual Pentium III 800 MHz, 512 MB RAM average error 7 %, median error 3 %

### 3.2 Incorporating Background Load

Besides the mode of operation, the observed performance of a broker also varies with the background load. It has been suggested [9] that the background load needs to be incorporated into the benchmark to make it relevant. Measurements done with a regular processor-intensive and network-intensive load created by background processes suggest that the background

load exhibits itself as an increase in the invocation times that is inversely proportional to the percentage of the processing power scheduled to the benchmarks (Figure 10).



**Figure 10** Roundtrip time for growing sequence<octet>, direction out, static invocation, **ORBacus 4.0.3**, Linux 2.2.18, Dual Pentium III 800 MHz, 512 MB RAM, lines for different numbers of background processes

The effects of an irregular background load cannot be incorporated into the benchmark results as easily. The simplest way to obtain precise results in irregular background load conditions is running the benchmark suite under these conditions.

### 3.3 Incorporating Network Conditions

Under congested network conditions, the influence of the particular broker implementation on the benchmark results tends to be quickly overshadowed by the influence of the network delays. Because a majority of brokers communicates through IIOP on top of TCP/IP implemented by the operating system protocol stack, it is best to measure their behavior under congested network conditions using the existing methodology for TCP/IP benchmarking.

#### **Typecode Indirection**

**Desc:** The test measures the typecode indirection mechanism that prevents duplication of typecodes in GIOP messages. The called interface is:

```
typedef any ArrayAny [100];
interface TypeCodeIndirection {
    void InArrayAny (in ArrayAny Arr);
};
```

The array is filled with instances of types whose typecodes are 100 characters long. The dependency of the invocation time on the number of identical types within the array is measured.

**Graph:** The graph depicts the dependency of the invocation time on the number of identical typecodes passed. The X axis gives the number of instances passed within the array that have the same type; the Y axis gives the invocation time.

**Result:** When no dependency is observed, the ORB does not use the typecode indirection mechanism. A gradually decreasing dependency should be exhibited otherwise.

**Figure 11** A report for a typecode benchmark

### 3.4 Reporting Results

The report with the results of the benchmarking suite should follow the proven full disclosure approach [22][23] by specifying configuration information that is sufficient to reproduce the results. Besides the results themselves, the report should contain a description of the benchmark, a description of the form in which the results are reported, an explanation of

what the results signify and when simplifications might not apply, for each benchmark. An example of a report for a single benchmark is on Figure 11.

#### 4 PERFORMANCE OF C++ ORBS

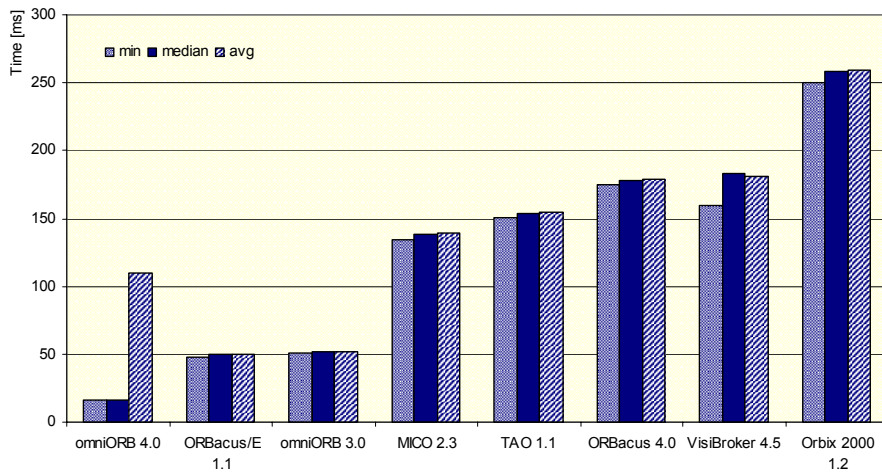
Although the presented benchmarking methodology is general enough, we have implemented it and use it only for C++ CORBA brokers. As we have collected many results for several C++ CORBA brokers, in this section we present the performance of several common brokers. Currently we are working on Java version benchmark suite.

There are less than ten generally known C++ brokers on market today. Unfortunately we did not obtain the licenses to test all of them. However, we have evaluated the most common brokers, which are:

- **MICO** – open-source broker; does not support multithreading.
- **omniORB** – open-source lightweight high-performance broker developed originally in Olivetti & Oracle Research Lab, later in AT&T Laboratories Cambridge, now without industrial support.
- **ORBacus, ORBacus/E** – open-source broker originally from OOC, now from IONA. Even though open-source, it is full-featured broker with couple of CORBA services. ORBacus/E was a version for embedded applications.
- **Orbix, Orbix/E** – well-known commercial general-purpose broker from IONA. Orbix/E is a lightweight version for usage in embedded applications, derived from ORBacus/E.
- **TAO** – open-source real-time high performance broker developed in the group of prof. Schmidt. Full feature broker with several CORBA services available.
- **VisiBroker** – well-known commercial general-purpose broker from Borland.

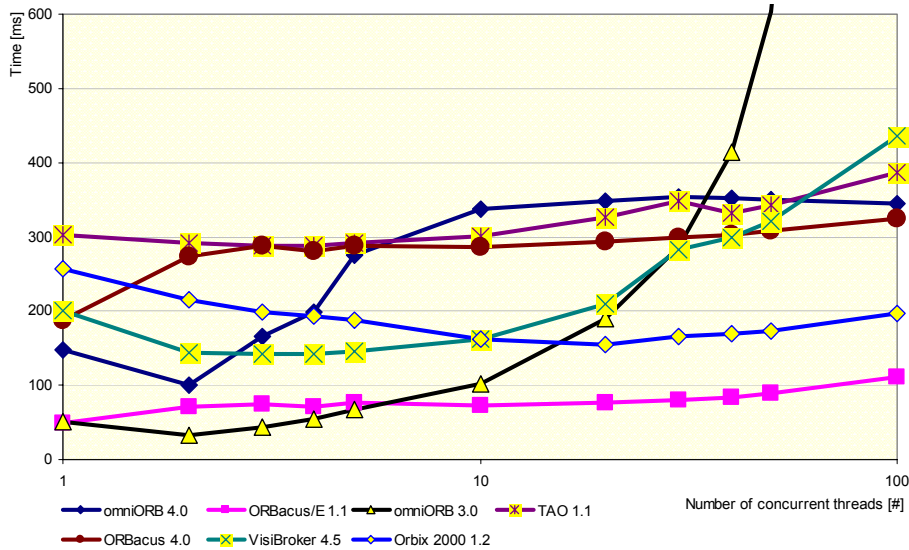
In the remaining of the section, we present several benchmarks and compare the results.

The Figure 12 shows the basic invocation time – roundtrip time for single empty call. The brokers are sorted according to median invocation time. Note the omniORB 4.0 difference between median and average time. It is caused by a couple of invocations that took extraordinary time to complete.



**Figure 12** Roundtrip time, simple invocation, static invocation, Linux 2.4.x, Dual Intel Pentium III 800 MHz, 512 MB RAM

The result of parallelism benchmark for all the tested brokers is shown on Figure 13. The typical curve is usually slightly better for certain number of concurrent threads (see Orbix 2000 or VisiBroker), but there are some implementations that do not follow this rule (see ORBacus 4.0).



**Figure 13** Average roundtrip time, void ping() for increasing number of threads, static invocation, Linux 2.4.x, Dual Intel Pentium III 800 MHz, 512 MB RAM

## 5 CONCLUSION

The simple benchmark suite outlined in section 2, provides an overview of most significant factors influencing the broker performance. The approach is used in benchmarking suite available at the web page <http://nenya.ms.mff.cuni.cz/~bench>. It is the first and only attempt to create such a library of benchmarks and database of results. In line with the suggestions of the OMG White Paper on Benchmarking [10], the suite is easy to execute and easy to port to different systems. It is also relevant in the sense outlined in section 3, because its results can be adjusted to reflect a particular mode of operation, and provides a good understanding of the broker performance issues. The suite has been implemented for C++ brokers.

An approach to porting the benchmark results has been suggested and, after tests on a wide range of systems, shown to provide very precise performance estimates. Using this approach, the benchmark results can be used to deduce system requirements from performance requirements, to compare the performance of brokers running at different systems, and other purposes that a usual benchmark does not lend itself to very well. For a future work, the approach should be verified on other than C++ platforms.

To conclude, we show that it is possible to generalize CORBA broker benchmarks into a relatively small suite, whose results can be tailored to a specific system and mode of operation without a prohibitive loss of precision.

## 6 ACKNOWLEDGEMENTS

The authors would like to thank members of the Distributed Systems Research Group at Charles University, headed by professor František Plášil, for cooperation on the various benchmarking projects, and the industrial partners in these projects, for providing feedback and allowing us to use the results for research purposes.

## 7 REFERENCES

- [1] Callison, H.R., Butler, D.G., Real-time CORBA Trade Study, Boeing, 2000
- [2] OMEX , CORBA Testbed, <http://www.omex.ch/resources/CorbaTB/corbatb.htm>, 1999
- [3] Amar, V., CORBA Benchmarks Results, <http://www.beust.com/virginie/Benchmarks>, 1999
- [4] Distributed Systems Research Group, CORBA Comparison Project, Final Report, Charles University, Prague, 1998
- [5] ORBOS Platform Task Force, Benchmark RFI, OMG Document bench/98-05-01, 1998
- [6] Plášil, P., Tůma, P., Buble, A., CORBA Benchmarking, Tech. Report No. 98/7, Dep. of SW Engineering, Charles University, 1998
- [7] MITRE/Open Systems Center, Proposal to the ORBOS Platform Task Force for Benchmarking CORBA Scalability, OMG document bench/98-10-01, 1998
- [8] Chung, M., France Telecom CNET Response to Benchmark RFI, OMG document bench/98-10-05, 1998

- [9] University of Helsinki, University of Helsinki Response to ORBOS Platform Task Force Benchmark RFI, OMG document bench/98-10-03, 1998
- [10] Object Management Group, White Paper on Benchmarking, Version 1.0, OMG document bench/99-12-01, 1999
- [11] Procházka, M., Tůma, P., Pospíšil, R., Enterprise JavaBeans Benchmarking, Tech. Report No. 4/2000, Dep. of SW Engineering, Charles University, 2000
- [12] Distributed Systems Research Group, EJB Comparison Project, Final Report Public Distribution Version, Charles University, Prague, 2000
- [13] Distributed Systems Research Group, CORBA Comparison Project Extension, Final Report, Charles University, Prague, 1999
- [14] Schmidt, D.C., Gokhale, A., Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks, IEEE 17th International Conference on Distributed Systems (ICDCS 97), 1997
- [15] Tůma, P., Buble, A., Open CORBA Benchmarking, Proceedings of SPECTS 2001, Orlando, FL, 2001
- [16] Tůma, P., Buble, A., Technical Report on Open CORBA Benchmarking, Tech Report No. 2001/1, Dep. of SW Engineering, Charles University, Prague, 2001
- [17] CORBA 3.0, OMG document formal/2002-06-33, 2002
- [18] Schmidt, D.C., Vinoski, S., An Overview of the OMG CORBA Messaging Quality of Service (QoS) Framework, C++ Report, SIGS, Vol. 12, No 3, March, 2000
- [19] Schmidt, D. C., Evaluating Architectures for Multi-threaded CORBA Object Request Brokers, CACM Vol. 41, No. 10, 1998
- [20] Mogul, J., Brittle Metrics in Operating Systems Research, Proceedings of the Seventh Workshop on Hot Topics in Operating Systems, 1998
- [21] Henning, J.L., SPEC CPU2000: Measuring CPU Performance in the New Millennium, IEEE/COMPUTER, July 2000
- [22] SPEC, SPEC CPU2000 V1.1 Documentation, <http://www.spec.org/osg/cpu2000/docs/>, 2000
- [23] TPC, Transaction Processing Performance Council Policies, [http://www.tpc.org/organizational\\_documents/TPC\\_Policies\\_v5.0.htm](http://www.tpc.org/organizational_documents/TPC_Policies_v5.0.htm), 2000