Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms

Christophe Demarey - Gael Harbonnier - Romain Rouvoy - Philippe Merle
Jacquard INRIA Project
Laboratoire d'Informatique Fondamentale de Lille
UMR CNRS 8022 / Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq Cedex, France
{Christophe.Demarey,Gael.Harbonnier,Romain.Rouvoy,Philippe.Merle}@lifl.fr

ABSTRACT

Nowadays, distributed Java-based applications could be built on top of a plethora of middleware technologies such as Object Request Brokers (ORB) like CORBA and Java RMI, Web Services, and component-oriented platforms like EJB or CCM. Choosing the right technology fitting with application requirements is driven by various criteria such as economic costs, available features, performance, etc.

The main contribution of this paper is to present an experience report on the design and implementation of a simple benchmark to evaluate the round-trip latency of various Java-based middleware platforms. Empirical results and analysis are discussed on a large set of widely available implementations including various ORB (Java RMI, Java IDL, ORBacus, JacORB, OpenORB, and Ice), Web Services projects (Apache XML-RPC and Axis), and component-oriented platforms (JBoss, JOnAS, OpenCCM, Fractal, ProActive).

Keywords

benchmarking, round-trip latency, Java-based middleware, ORB, CORBA, Web services, EJB, CCM.

1. INTRODUCTION

Nowadays, distributed Java-based applications could be built on top of a plethora of middleware technologies such as Object Request Brokers (ORB), Web Services, and component-oriented platforms [39]. On one hand, an ORB mainly provides a middleware layer for transporting method invocations between distributed objects as addressed by the Object Management Group's Common Object Request Broker Architecture (OMG CORBA) [37, 3] and the Sun Microsystems's Java Remote Method Invocation (Java RMI) [29] specifications. One of the main non technical advantages of CORBA versus Java RMI is that it is an open vendor-neutral specification and then a lot of implementations are available like Sun's Java IDL [27], IONA's ORBacus [41], JacORB [40], OpenORB [38], etc. instead of only one for Java RMI [30]. Nevertheless, non-standard ORB are still designed for studying and providing new features and optimizations. For instance, the ZeroC's Ice ORB [19] is a new object-oriented middleware platform [15] similar in concept to CORBA but both simpler and more powerful for building large scale real-time distributed applications, like massively multiplayer online games [16]. On the other hand, Web Services are an alternative to ORB as they provide a similar transport layer for remote calls between heterogenous distributed services. The service requests are transported by standard Internet protocols and are encoded into XML documents as specified by the World Wide Web Consortium (W3C)'s Simple Object Access Protocol (SOAP) recommendation [43].

However, transparent remote interactions are not enough for building complex business applications where deployment, life cycle, security, transactions, and persistence are some examples of system aspects to be taken into account by designers. Then, component-oriented platforms, built on top of ORB, provide a container layer encapsulating business code and dealing with system aspects transparently as defined in the Sun's Enterprise JavaBeans (EJB) [8] and the OMG's CORBA Component Model (CCM) [36, 44] specifications. The main non technical advantage of EJB versus CCM is that EJB is implemented by a large set of commercial and open source products like IBM's WebSphere [17], JBoss [10, 18], or JOnAS [32] when CCM is only implemented by few open source projects like our OpenCCM platform [33]. Beside these standards, many academic researches are still done on new component-oriented middleware platforms. For instance, the Fractal project [35] proposes a new hierarchical, reflective, extensible, and efficient component model with sharing [4]. This model is extended in the ProActive project [20] for supporting Grid computing applications [1].

Choosing the right middleware technology fitting with application requirements is a complex activity as it is driven by a plethora of criteria such as economic costs (e.g. commercial or open source availability, engineer training and skills), conformance to standards, advanced proprietary features, performance, scalability, etc. Regarding performance, a lot of basic metrics could be evaluated like round-trip latency, jitter, or throughput of twoway interactions according

to various parameter types and sizes. Many projects have already evaluated these middleware performance metrics by benchmarking Java RMI versus CORBA [24, 22, 23], and various implementations of CORBA [13, 6, 42], EJB [14, 7, 26], or CCM [25]. Their results are very relevant for application developers who want to select the best implementation of an already selected kind of middleware technologies. Unfortunately, no past project has compared different kinds of middleware platforms simultaneously. This could be helpful for application designers requiring to select both the kind of middleware technology to apply and the best implementation to use.

The main contribution of this paper is to present an experience report on the design and implementation of a simple benchmark to evaluate the round-trip latency of various Java-based middleware platforms, i.e. only measuring the response time of twoway interactions without parameters. Even if simple, this benchmark is relevant as it allows users to evaluate the minimal response mean time and the maximal number of interactions per second provided by a middleware platform. For this purpose, empirical results and analysis are discussed on a large set of widely available Java-based middleware technologies including various implementations of ORB (Java RMI, Java IDL, ORBacus, JacORB, OpenORB, and Ice), Web Services (Apache XML-RPC and Axis), and component-oriented platforms (JBoss, JOnAS, OpenCCM, Fractal, ProActive).

The remainder of this paper is organized as follows. Section 2 gives an overview of our round-trip latency benchmark. Section 3 analyses preliminary empirical results obtained on a large set of benchmarked middleware platforms. Section 4 describes some related work on middleware benchmarking. Section 5 presents concluding remarks.

2. OUR ROUND-TRIP LATENCY BENCHMARK

This section gives an overview of our simple round-trip latency benchmark, outlines its main benchmarking objectives, identifies the key benchmarking challenges to resolve, presents the scenario of the benchmark and its associated configuration parameters.

2.1 The Benchmark Objectives

The design of our round-trip latency benchmark was driven by the following objectives:

Benchmarking heterogeneous Java-based midddleware platforms: Ideally, software designers want to build their distributed applications independently of any middleware platform and deploy them on various platforms. Model Driven Software Engineering (MDSE) approaches like the OMG's Model Driven Architecture (MDA) [31] address this by allowing us to design platform independent application models and map them to various middleware platforms automatically. In this context, benchmarking various heterogeneous Java-based midddleware platforms simultaneously is very crucial in order to be able to compare them and select the right one according to performance requirements of targetted applications.

Evaluating the best round-trip latency: A lot of benchmarking measures could be done according to taken metrics and their configuration parameters. In a preliminary step, our project only focusses on the evaluation of the best round-trip latency provided by various Java-based middleware platforms. Even if this metric is very simple, it provides a relevant overview of the best performance provided by each platform to distributed applications.

Comparing various Java-based CORBA implementations: The CORBA specification is designed to allow portability of applications on top of different CORBA products. For instance, our OpenCCM project, providing an open source CCM implementation, could be built and run on top of any Java-based CORBA compliant platform. Then comparing various Java-based CORBA implementations helps any CORBA-based software designers/users to select the best implementation in order to deploy and run their applications.

Comparing CORBA/IIOP versus other ORB protocols: Each middleware platform provides at least a transport protocol of remote interactions between distributed entities (i.e. objects, services or components). These protocols encompass rules for encoding interactions and data types, and for using underlying network protocols. In the CORBA specification, the General Inter-Orb Protocol (GIOP) defines encoding rules and the Internet Inter-Orb Protocol (IIOP) makes use TCP/IP for transporting object requests. The Java RMI platform allows applications to use both IIOP and the proprietary Java Remote Method Protocol (JRMP). However, some middleware platforms like Fractal and Ice provide their own remote method invocation protocols simpler and supposedly more optimized and efficient than GIOP/IIOP. Then comparing CORBA/IIOP versus other ORB protocols is important as the used transport protocol strongly impacts the global performance when many distributed entities interact together.

Evaluating XML-based middleware overhead: For most of industrial users and vendors, Web Services are more and more seen as an alternative to ORB as they provide also a transport layer between distributed heterogenous software services. The W3C has started to standardize this layer via the Simple Object Access Protocol (SOAP) recommendation. Nevertheless, SOAP certainly introduces some overhead regarding optimized ORB platforms due to high costs for parsing requests encoded in XML documents and dispatching them via HTTP servers.

Evaluating container overhead: In component-oriented middleware platforms, business code is encapsulated into containers dealing with some system services like life cycle, security, transactions, and persistence transparently. This container layer is built on top of ORB in order to inherit from distributed communication facilities, e.g. EJB and CCM platforms are built on top of Java RMI and CORBA implementations. As containers introduce an intermediate layer between ORB and business code, it is interesting to evaluate the overhead added when measuring the round-trip latency. However, this evaluation should be careful of deactivating services dealt with by the container as for instance the propagation of security credentials, the security access control, and the propagation and demarcation of transactions.

Providing a reusable benchmark software: The results produced by any benchmark strongly depend on the used hardware and software platforms. Moreover, benchmarking Java-based middleware also depends on the used operating system, the Java Virtual Machine, the version of the middleware platform, and the middleware configuration

(log levels, size of various pools, threading policies, etc.). Then our last objective is to develop a reusable benchmark software that could be applied directly by users to evaluate middleware platforms on their specific hardware and software platforms.

2.2 Benchmarking Challenges

During the design of our benchmark, we encountered the following challenges: heterogeneity in middleware, distributed execution, cold start issues, garbage collection perturbation, and large amount of measures. We describe each of these challenges below and outline how we are addressing these challenges.

2.2.1 Heterogeneity in Middleware

Our first challenge in developing the benchmark stemmed from the heterogeneity in the tools and mechanisms used in different middleware platforms in terms of means for:

- **Describing remote interactions**: According to the used middleware platform, different languages must be used to describe the public remote methods offered by a piece of software, e.g. 1) Java interfaces for Java RMI objects, EJB, Fractal, and ProActive components, 2) OMG IDL for CORBA objects and components, 3) XML for Web Services, or 4) Ice IDL for Ice objects. Moreover even with CORBA, each OMG IDL compiler has different executable name and command line options.
- **Implementing benchmark code**: As each middleware platform provides its own programming model, it is impossible to implement benchmark code in a portable way, e.g. the Java code for benchmarking Java RMI is strongly different from the code for CORBA, Ice, Web Services, Fractal, or ProActive. Fortunately as EJB and CORBA are specifications, the associated benchmarking code could be written in a portable way independently of underlying platform implementations, i.e. the same code is used for benchmarking all CORBA platforms and another one is used for all EJB platforms.
- **Deploying the benchmark**: Deploying the benchmark is also dependent on the used middleware platform, e.g. a different set of JAR archives containing the platform runtime, different scripts to start needed platform services, different formalisms to describe EJB, CCM, Fractal, and ProActive components to deploy.

In order to address the challenge of heterogeneity in middleware, we have structured the benchmark software into several modules, i.e. one for each heterogenous Java-based middleware technology (javarmi, corba, ice, xml-rpc, axis, ejb, openccm, fractalrmi, proactive). Each module contains a set of Ant, Java, configuration, and script files specific to the benchmarked middleware technology. Fortunately, standardized technologies like CORBA and EJB allowed us to factorize most of the files common to different platform implementations, and to only provide some configuration files specific to each benchmarked implementation. However, our current approach would be replaced by a MDSE approach where all files needed for implementation, compilation, and deployment could be generated automatically from a common platform independent model (PIM).

2.2.2 Distributed Execution

When benchmarking middleware, we must run distributed applications implying distributed synchronization issues. For instance, client processes could be started only when server processes are completely initialized. We currently address this issue via a distributed barrier mechanism which guarantees that the benchmark is started only when servers are ready. However, this ad hoc mechanism would be replaced by the use of a general benchmarking platform that could provide generic and automatic mechanisms for distributed deployment, execution and synchronization as targetted for instance by the CLIF platform [9, 34].

2.2.3 Cold Start Issues

During the evaluation of various Java-based middleware platforms, we have encountered various cold start issues or warm-up effects as discussed in [5]. For example, each middleware platform implements caches to deal with network channels, request buffers, and threads. Moreover, modern JVMs provide Just In Time (JIT) compilation mechanisms to transform Java bytecodes to machine code for improving performances. All these mechanisms become fully efficient after a given number of interactions, this number depending on the benchmarked software platform (JVM + middleware). In order to benchmark the best round-trip latency of a middleware platform, it is necessary to start measures only after these mechanisms are fully activated. Then as recommended in [5], our benchmark always executes a large set of preliminary first interactions before measuring next interactions.

2.2.4 Garbage Collection Perturbation

By default, Java provides a garbage collection (GC) mechanism which automatically destroys objects which are not referenced by applications yet. As this GC mechanism is activated in a non deterministic way, this introduces perturbations when measuring latency. However, without automatic GC, most of Java-based middleware platforms could not be run during a long time as they will consume more than available memory resources. Then we decide to introduce a parameter to the benchmark allowing the activation/deactivation of the GC, and we activate GC by default in order to evaluate the common use of Java-based middleware platforms.

2.2.5 Large Amount of Measures

Benchmarking various Java-based middleware platforms requires to collect and analyse a large amount of measures due to the plethora of middleware platforms to evaluate, the determination of the number of first interactions to execute for avoiding cold start issues, and the perturbations introduced by garbage collection. Currently, we use text files for collecting this large amount of measures and use spreadsheets for analysing them. However, this simple approach would be replaced by a more advanced benchmarking platform managing a database containing all benchmarking scenario descriptions, conditions and effective measures. This database would be accessed from the Web to query various analysis as it is already done in the Open CORBA Benchmarking project [42].

2.3 The Benchmark Scenario

Our round-trip latency benchmark measures round-trip response times of simple twoway interactions with no argument and a void return value, i.e. the public synopsis of interactions is *void ping()*. The benchmark is composed of two applications running into two different JVMs: The server provides a resource (object, service or component) implementing an empty *ping* method and is invoked by the remote client. Each interaction is marshalled by a client stub and propagated to a server skeleton through the transport layer provided by the middleware platform. This request is unmarshalled by the skeleton and the method implementation is invoked. Then a void reply is sent back from the skeleton to the client using the same mechanism.

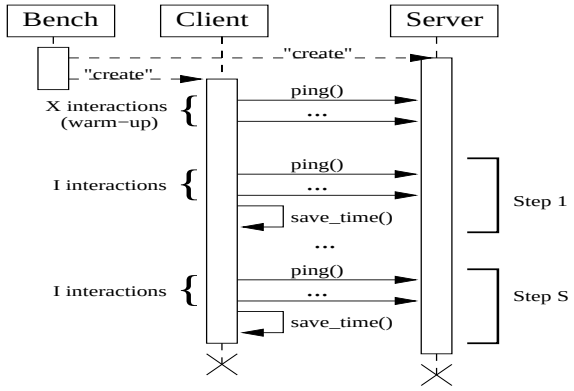

Figure 1: The sequence diagram for all series.

Our benchmark is divided into N series executed sequentially. As shown in Figure 1, all series are made of a server starting, a client starting and a server shutdown. The client application performs X first interactions in order to warm-up the whole benchmarked system, e.g. transport layer initialization (socket creation), cache mechanism startup at ORB and container levels, and JIT activation. Then S steps of I interactions are performed. The global time of each step is collected via the `System.currentTimeMillis` method from Java SDK, and an average measure for one interaction is stored. This is useful to observe the evolution of round-trip measures according to steps. Moreover, this scenario can be played with two JVMs on one or two hosts in order to respectively avoid or measure the network impact. JVMs can be configured to activate or deactivate JIT and GC mechanisms separately.

3. EMPIRICAL RESULTS

This section presents some empirical results of our round-trip latency benchmark executed on a large set of widely available Java-based middleware platforms. Firstly, we describe the hardware and software platforms and the benchmark configuration commonly used for all benchmarked middleware platforms. Then, platform evaluations are presented and discussed by middleware category: ORB, Web Services, and component-oriented platforms.

3.1 Hardware and Software Platforms

The experiments presented in this section were conducted using the same Dell Optiplex GX 240 workstation with a single Intel Pentium 4 processor (2 GHz) and 1 GB of RAM. The operating system is Linux Debian with a minimal 2.4.18-1 kernel (i686 package) and without X server. All experiments were performed on the same Java Virtual Machine, i.e. the Sun Microsystems's JDK 1.4.2-b28.

Benchmarked Java-based ORB platforms are:

- Java Remote Method Invocation version 1.4.2 [30, 28] over both JRMP and IIOP protocols.
- Java IDL version 1.4.2 [27] - The Sun's CORBA implementation.
- ORBacus version 4.1.0 [41] - The IONA's commercial CORBA implementation.
- JacORB version 2.1 and 2.2 [2] - An open source CORBA implementation.
- The Community OpenORB version 1.3.1 and 1.4.0 [38] - An open source CORBA implementation.
- Ice 1.5.0 [19] - The ZeroC's proprietary, optimized, and efficient Internet Communications Engine (Ice) object-oriented platform.

Benchmarked Java-based Web Services platforms are:

- Apache XML RPC version 1.1 [11] - A free Java implementation of XML-RPC, a popular protocol that uses XML over HTTP to implement remote procedure calls.
- Apache Axis version 1.1 [12] - An open source implementation of the W3C's SOAP 1.2 recommendation.

Benchmarked Java-based component-oriented platforms are:

- JBoss version 4.0.0RC1 [18] - The famous world wide known open source J2EE implementation.
- JOnAS version 4.1.2 [32] - The ObjectWeb's open source J2EE implementation.
- OpenCCM version 0.8.1 [33] - Our ObjectWeb's open source CCM implementation evaluated on top of ORBacus 4.1.0, JacORB 2.1/2.2, and OpenORB 1.3.1/1.4.0.
- Fractal RMI version 0.3 [35] - The ObjectWeb's proprietary reflective and extensible component model.
- ProActive version 2.0 [20] - An INRIA's proprietary library designed for parallel, distributed, and concurrent grid computing.

3.2 The Benchmark Configuration

For all evaluated middleware platforms, our benchmark scenario is configurated as follows. The server and client run into two JVMs on the same host to avoid network perturbations. JIT and GC are activated to obtain the best performance and to evaluate the common use case of Java-based middleware platforms respectively. 50 benchmark series are executed sequentially. For all series, 10000 first invocations are performed and not measured to remove most of the cold start issues. Then 20 steps of 500 interactions are measured. This benchmark configuration (N=50, X=10000, S=20, I=500) is significant as 1000 measures (50 series of 20 steps) are obtained for each benchmarked middleware platform, representing half a million of measured interactions.

Middleware	Interactions per second	Mean Time in us	Min Time in us	Max Time in us	Standard Deviation
Java RMI/JRMP 1.4.2	8088.78	123.63	108	196	9.86
Java RMI/IIOP 1.4.2	1932.94	517.35	460	624	25.01
Java IDL 1.4.2	1566.89	638.21	586	714	22.32
ORBacus 4.1.0	8690.06	115.07	96	186	11.59
JacORB 2.1	3849.14	259.80	164	430	49.63
JacORB 2.2	1799.03	555.86	496	736	35.20
OpenORB 1.3.1	1762.40	567.41	490	828	47.71
OpenORB 1.4.0	1578.48	633.52	504	886	60.11
ICE 1.5.0	2960.91	337.73	234	518	68.06
XML-RPC 1.1	640.18	1562.07	1508	1646	33.82
Axis 1.1	210.85	4742.70	4556	13334	742.89
JBoss 4.0.0RC1	744.96	1342.36	1214	2062	133.23
JOnAS 4.1.2	3670.59	272.44	246	352	15.30
OpenCCM over ORBacus 4.1.0	5374.90	186.05	150	340	17.04
OpenCCM over JacORB 2.1	2433.13	410.99	272	780	91.96
OpenCCM over JacORB 2.2	1374.40	727.59	656	1014	44.09
OpenCCM over OpenORB 1.3.1	1190.07	840.29	724	1074	102.83
OpenCCM over OpenORB 1.4.0	1100.10	909.01	764	1168	109.54
Fractal RMI 0.3	6601.27	151.49	104	262	21.37
ProActive 2.0	482.90	2070.83	2010	2384	38.26

Table 1: Round-trip latency benchmark results of various Java-based middleware platforms.

Table 1 shows round-trip latency measure results for each benchmarked middleware platform including 1) the maximal number of interactions per second provided by a platform, 2) the mean, minimal, and maximal response time in micro seconds for one interaction, and 3) the standard deviation. These results are discussed in next sections by middleware category.

3.3 Object Request Brokers

Table 1 clearly shows that ORBacus 4.1.0 and Java RMI/JRMP 1.4.2 are the two most efficient and stable benchmarked ORB as they provide similar round-trip latency results. This is not a surprise because these two ORB have been developed, optimized, and used in many applications over the world for several years. The overhead factors introduced by other benchmarked ORB are depicted in Table 2.

ORB	Vs RMI/JRMP	Vs ORBacus
ORBacus 4.1.0	*0.93	
Java RMI/JRMP 1.4.2		*1.07
JacORB 2.1	*2.10	*2.26
ICE 1.5.0	*2.73	*2.93
Java RMI/IIOP 1.4.2	*4.18	*4.50
JacORB 2.2	*4.50	*4.83
OpenORB 1.3.1	*4.59	*4.93
OpenORB 1.4.0	*5.12	*5.51
Java IDL 1.4.2	*5.16	*5.55

Table 2: ORB round-trip latency compared to Java RMI/JRMP 1.4.2 and ORBacus 4.1.0.

Regarding Sun's platforms only, it is clear that the Sun's IIOP implementation is less efficient than JRMP (by a factor near to 4). Then applications should use Java RMI/JRMP instead of Java RMI/IIOP and Java IDL. Regarding CORBA/IIOP only, benchmarked Java IDL, JacORB, and OpenORB platforms introduce an important overhead factor between 2 and 5 compared to ORBacus 4.1.0. Moreover, we could observe benchmarking regression for some platforms, e.g. JacORB 2.2 is two times less efficient than JacORB 2.1. This is mainly due to the fact that these platforms are still in development and their ORB core is strongly redesigned between releases. We could also conclude that Java IDL 1.4.2 provides the worst results compared to any other benchmarked CORBA platforms. Finally, our results show that the recent Ice platform does not bring the performance and efficiency promised by its authors in [15] as it is less efficient than ORBacus 4.1.0, Java RMI/JRMP 1.4.2, and JacORB 2.1.

3.4 Web Services

As depicted in Table 1, benchmarked Web Services platforms introduce a considerable overhead compared to any benchmarked ORB platform. In the worst case, Apache XML-RPC 1.1 and Axis 1.1 are near to be 13.5 and 41 times slower than ORBacus 4.1.0. This is mainly due to the fact that they use XML and HTTP to respectively encode and transport service requests when CORBA with GIOP/IIOP provides a compact binary transport protocol built on top of TCP/IP directly. Moreover, XML-RPC 1.1 is 3 times faster than Axis 1.1 because it implements a RPC protocol simpler that SOAP regarding both encoding and transport rules. Nevertheless, better results should be obtained by using optimized XML parsers and encoding XML via binary formats. However, these will still require to improve performance by a high factor at least superior to 10. Then, the benchmarked Web Services platforms could not be used when applications do require a high number of remote interactions or near real-time properties.

Component Middleware	Underlying ORB	Mean Time in us	Overhead in us	Overhead in %
OpenCCM 0.8.1	ORBacus 4.1.0	186.05	70.98	162%
JOnAS 4.1.2	Java RMI/JRMP 1.4.2	272.44	148.81	220%
OpenCCM 0.8.1	JacORB 2.1	410.99	151.20	158%
OpenCCM 0.8.1	JacORB 2.2	727.59	171.14	131%
OpenCCM 0.8.1	OpenORB 1.3.1	840.29	272.88	148%
OpenCCM 0.8.1	OpenORB 1.4.0	909.01	275.49	143%
JBoss 4.0.0RC1	Java RMI/JRMP 1.4.2	1342.36	1218.73	1086%
ProActive 2.0	Java RMI/JRMP 1.4.2	2070.83	1947.20	1675%

Table 3: Component middleware overhead added to underlying ORB.

3.5 Component-Oriented Platforms

In order to compare the various benchmarked component-oriented platforms, the server component is designed as a stateless session component hosted by a minimalist container, e.g. with no security and transaction propagation and management. For distributed communication, both EJB implementations and ProActive are configurated to use Java RMI over JRMP, OpenCCM uses one of underlying CORBA implementations, and Fractal uses its own proprietary component-based ORB.

As shown in Table 1, the Fractal RMI platform brings the best latency results compared to any other benchmarked component platforms. These excellent results are close to those of most efficient ORB mainly because this platform implements an efficient component model as promised in [4] and its own ORB provides a minimalist RPC network protocol simpler than JRMP and IIOP. On the other hand, ProActive 2.0, another implementation of the Fractal component model dedicated to grid computing, does not yet bring excellent results. These worst results are certainly due to the fact that ProActive is based on a Meta Object Protocol (MOP) [1] using non optimised reflective technics intensely.

As shown in Table 3, the round-trip latency provided by component-oriented platforms is strongly dependent on the underlying used ORB, i.e. more the underlying ORB is performant more the component platform is performant. Especially, this could be observed with the OpenCCM platform where the same container code is run on top of different CORBA platforms and uses Portable Object Adapters (POA) intensely. The global OpenCCM latency is the sum of the latency of the underlying CORBA implementations (see Table 2), the latency of underlying POA, and the latency of the OpenCCM container (always constant as the same code is run). Then firstly, the ranks of OpenCCM latency results are similar to those obtained with underlying CORBA platforms only, i.e. overall placings are ORBacus 4.1.0, JacORB 2.1, JacORB 2.2, OpenORB 1.3.1, and OpenORB 1.4.1. Secondly, the overhead introduced by the OpenCCM container fully depends on how the underlying ORB implements the POA. Moreover, we could observe that OpenCCM 0.8.1 over ORBacus 4.1.0 provides better latency results and introduces a smaller container overhead than JOnAS 4.1.2 and JBoss 4.0.0RC1 over Java RMI/JRMP 1.4.2: This is a first encouraging result for our OpenCCM platform.

However, we could observe that the benchmarked component platforms, excepts Fractal RMI, introduce some significant container overhead regarding to underlying ORB: from +31% to +62% for OpenCCM, +120% for JOnAS, 1086% for JBoss, and 1675% for ProActive. Then we argue that a lot of research works still must be done in order to strongly optimize these component platforms. As shown by the Fractal project, an efficient component platform could minimalize the container overhead and provide latency results near to efficient pure ORB. According to the first encouraging results on our OpenCCM platform, our future work will target to identify container bottlenecks, and to propose and validate container optimizations.

4. RELATED WORK

Middleware benchmarking was, is, and will always be a very hot topic for the middleware community as this allows us to evaluate and compare platforms in order to select the best one according to application requirements, to identify bottlenecks in implementations, and to propose and validate optimizations. Then a lot of past middleware benchmarking projects have already evaluated a large set of performance metrics (not just round-trip latency) and proposed relatively complete benchmarking frameworks. But unfortunately each of them only focusses on a particular middleware technology and could not provide a global overview as our benchmark does.

Regarding Sun's ORB platforms, Juric and al. have strongly evaluated and compared Java RMI/JRMP, Java RMI/IIOP, and Java IDL round-trip latency for remote method calls with various parameter types and sizes [21, 24, 22, 23]. In [23], they concluded that these three technologies in version 1.3 have similar performance results, but these results are obtained without warm-up and for 200 iterations only as pointed out by [5]. Then with significant warm-up (i.e. 10000) and number of iterations (i.e. half a million), our results show that Java RMI/JRMP 1.4.2 is considerably faster than both Java RMI/IIOP and Java IDL 1.4.2.

Regarding CORBA, various benchmarking platforms and results were published [13, 6, 42]. One of the most still active projects is the Open CORBA Benchmarking project [42] of the Distributed Systems Research Group at Charles University. This project provides several benchmark suites for evaluating various C++ and Java CORBA implementations, and a powerful Web-based repository to submit and query benchmark results. Even if few results for Java-based CORBA implementations are available in this repository, they confirm our empirical results: ORBacus is faster than OpenORB and Java IDL.

Evaluating the performance and scalability of J2EE application servers and containers is a hot active topic as discussed in depth in [14, 7] and specified by the Sun's ECperf specification [26]. Regarding CCM, only the CCMPerf benchmarking tool exists [25] but it just targets C++ based im-

plementations. In our work, we have evaluated few EJB and CCM platforms but our results show that Java-based containers introduce an important overhead when measuring the round-trip latency, and that a lot of works is still to be done in order to optimize them.

To the best of our knowledge, we are the first project that target to evaluate and compare various heterogeneous Java-based middleware platforms simultaneously. In return, we only benchmark the round-trip latency for simple ping interactions, and do not provide as much performance metrics as other benchmarking projects provide. However, this work provides a global overview of the best performance that benchmarked platforms could provide.

5. CONCLUDING REMARKS

Nowadays, Model Driven Software Engineering (MDSE) like the OMG MDA promotes a novel model driven approach to design distributed applications and map them automatically on top of the plethora of existing middleware technologies such as Object Request Brokers (ORB) like CORBA and Java RMI, Web Services, and component-oriented platforms like EJB or CCM. In this context, benchmarking various heterogeneous midddleware platforms simultaneously is very crucial in order to be able to compare them and select the right one according to performance requirements of targetted applications. However, a lot of performance metrics could be evaluated like round-trip latency, jitter, or throughput of twoway interactions according to various parameter types and sizes.

The main contribution of this paper was to present an experience report on the design and implementation of a simple round-trip latency benchmark to evaluate various heterogeneous Java-based middleware platforms. Even if simple, this benchmark is relevant as it evaluates the minimal response mean time and the maximal number of interactions per second provided by a middleware platform. For this purpose, empirical results and analysis were discussed on a large set of widely available implementations including various ORB (Java RMI, Java IDL, ORBacus, JacORB, OpenORB, and Ice), Web Services projects (Apache XML-RPC and Axis), and component-oriented platforms (JBoss, JOnAS, OpenCCM, Fractal, ProActive).

Regarding to our objectives defined in Section 2.1, following concluding remarks could be expressed:

- Benchmarking heterogeneous Java-based middleware platforms implies to resolve the key challenge of heterogeneity in middleware. Then we argue for applying a MDSE approach in order to capture abstract benchmark scenarios and to automatically generate all implementation, compilation, and deployment artefacts required by targetted middleware.

- Evaluating the best round-trip latency of any Java-based middleware platform requires to take care about warm-up effects. Then it is needed to execute a large number of interactions (e.g. 10000) before measuring the round-trip latency of a large set of next interactions.

- Comparing various Java-based CORBA implementations has shown that ORBacus 4.1.0 is between 2 and 5 more efficient than JacORB 2.1/2.2, OpenORB 1.3.1/1.4.0 and Java IDL 1.4.2. Moreover Java IDL is the worse of benchmarked CORBA platforms.

- Comparing CORBA/IIOP versus other ORB protocols has shown that GIOP/IIOP implemented in an efficient way (i.e. in ORBacus) provides better round-trip latency results than other benchmarked ad hoc protocols as provided by Java RMI/JRMP, Ice, and Fractal RMI platforms.

- Evaluating XML-based middleware overhead has shown that benchmarked Web Services platforms introduce a high significant overhead compared to ORB due to high costs of XML parsing and HTTP routing. Then we argue that these Web Services platforms could not be an alternative to ORB when distributed services interact strongly together or for near real-time systems.

- Evaluating container overhead has shown that most of benchmarked component platforms introduce an important overhead in containers. The only exception is the Fractal RMI platform which provides efficient reflective technics to implement containers with minor overhead. Then we argue that a lot of research works is still needed in order to propose and validate container optimizations.

Regarding our last objective, the benchmark presented in this paper is freely available under the GNU LGPL license in the *benchmark* CVS module of the OpenCCM platform at `http://openccm.objectweb.org`.

6. REFERENCES

[1] F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - Proceedings of OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2003*, volume 2888 of *Lecture Notes in Computer Science (LNCS)*, pages 1226–1242, Catania, Sicily, Italy, November 3-7 2003. Springer-Verlag.

[2] G. Brose. JacORB Web Site. *http://www.jacorb.org*, 2003.

[3] G. Brose, A. Vogel, and K. Duddy. *Java Programming with CORBA, 3rd ed.* John Wiley and Sons, New York, USA, Jan. 2001.

[4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An Open Component Model and Its Support in Java. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering - CBSE7*, volume 3054 of *Lecture Notes in Computer Science (LNCS)*, pages 7–22, Edinburgh, Scotland, May 24-25 2004. Springer-Verlag.

[5] A. Buble, L. Bulej, and P. Tuma. CORBA Benchmarking: A Course With Hidden Obstacles. In *Proceedings of the IPDPS Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEOPDS 2003)*, pages 279–284, Nice, France, Apr. 2003. IEEE Computer Society.

[6] H. R. Callison and D. G. Butler. Real-time CORBA Trade Study. Technical report, Boeing, 2000.

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, pages 246–261, 2002.

[8] L. G. DeMichiel. Enterprise JavaBeans Specification. Version 2.1, Final Release, Sun Microsystems Inc., Nov. 2003. *http://java.sun.com/products/ejb/docs.html*.

[9] B. Dillenseger and E. Cecchet. CLIF is a Load Injection Framework. In *Proceedings of the OOPSLA 2003 Workshop on Middleware Benchmarking*, Anaheim, CA, USA, October 26 2003.

[10] M. Fleury and F. Reverbel. The JBoss Extensible Server. In *Middleware 2003 - ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *Lecture Notes in Computer Science (LNCS)*, pages 244–373, Rio de Janeiro, Brazil, June 2003. Springer-Verlag.

[11] T. A. S. Foundation. Apache XML-RPC. *http://ws.apache.org/xmlrpc/*, 2003.

[12] T. A. S. Foundation. WebServices - Axis. *http://ws.apache.org/axis/*, 2003.

[13] D. S. R. Group. CORBA Comparison Project. Project Extension Final Report, Charles University, Aug. 1999.

[14] D. S. R. Group. EJB Comparison Project. Final Project Report, Public Distribution Version, Charles University, Feb. 2000.

[15] M. Henning. A New Approach to Object-Oriented Middleware. *IEEE Internet Computing*, pages 66–75, January - February 2004.

[16] M. Henning. Massively Multiplayer Middleware. *ACM QUEUE*, pages 40–45, Feb. 2004.

[17] IBM. WebSphere Software Platform. *http://www-306.ibm.com/software/info1/websphere*, 2004.

[18] J. Inc. JBoss Web Site. *http://www.jboss.org*, 2004.

[19] Z. Inc. Ice -The Internet Communications Engine. *http://www.zeroc.com/ice.html*, 2004.

[20] INRIA. ProActive Web Site. *http://proactive.objectweb.org*, 2002.

[21] M. Juric, A. Zivkovic, and I. Rozman. Comparison of CORBA and Java RMI Based on Performance Analysis. In *Proceedings of MHSS'98*, 1998.

[22] M. B. Juric and I. Rozman. Java 2 RMI and IDL Comparison. *Java Report*, 5(2):36–48, Feb. 2000.

[23] M. B. Juric and I. Rozman. RMI, RMI-IIOP and IDL Performance Comparison. *Java Report*, 6(4), Apr. 2001.

[24] M. B. Juric, A. Zivkovic, and I. Rozman. Are Distributed Objects Fast Enough? *Java Report*, 3(5):29–38, May 1998.

[25] A. S. Krishna, J. Balasubramanian, A. Gokhale, D. C. Schmidt, D. Sevilla, and G. Thaker. Empirically Evaluating CORBA Component Model Implementations. In *Proceedings of the ACM OOPSLA 2003 Workshop on Middleware Benchmarking*, Anaheim, CA, USA, October 26 2003.

[26] S. Microsystems. ECperf Specification. Technical report, Sun Microsystems Inc., Apr. 2002. *http://java.sun.com/j2ee/ecperf/*.

[27] S. Microsystems. Java IDL Technology Documentation. *http://java.sun.com/j2se/1.4.2/docs/guide/idl/*, 2002.

[28] S. Microsystems. Java Remote Method Invocation (RMI) over IIOP Documentation. *http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/*, 2002.

[29] S. Microsystems. Java Remote Method Invocation Specification. Revision 1.8, Sun Microsystems Inc., 2002. *http://java.sun.com/j2se/1.4.2/docs/guide/rmi/*.

[30] S. Microsystems. Java Remote Method Invocation (RMI). *http://java.sun.com/j2se/1.4.2/docs/guide/rmi/*, 2003.

[31] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. OMG TC Document omg/03-03-01, Object Management Group, Needham, MA, USA, June 2003.

[32] ObjectWeb. JOnAS: Java Open Application Server. *http://jonas.objectweb.org*, 2002.

[33] ObjectWeb. OpenCCM - The Open CORBA Component Model Platform. *http://openccm.objectweb.org*, 2002.

[34] ObjectWeb. The CLIF Project. *http://clif.objectweb.org*, 2002.

[35] ObjectWeb. The Fractal Project. *http://fractal.objectweb.org*, 2002.

[36] OMG. CORBA Components Specification, Version 3.0. OMG TC Document formal/02-06-65, Object Management Group, Needham, MA, USA, June 2002.

[37] OMG. Common Object Request Broker Architecture: Core Specification, Version 3.0.3. OMG TC Document formal/04-03-12, Object Management Group, Needham, MA, USA, Mar. 2004.

[38] T. OpenORB Team. The Community OpenORB Project. *http://openorb.sourceforge.net*, 2002.

[39] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Westley, New York, USA, 2nd edition, Nov. 2002.

[40] T. J. Team. *JacORB 2.2 Programming Guide*, May 2004. *http://www.jacorb.org*.

[41] I. Technologies. ORBacus Web Site. *http://www.orbacus.com*, 2004.

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

[43] W3C. Simple Object Access Protocol (SOAP) Version 1.2. Recommendation, World Wide Web Consortium, 2004. *http://www.w3.org/TR/soap/*.

[44] N. Wang, D. Schmidt, and C. O'Ryan. *Component-Based Software Engineering - Putting the Pieces Together*, chapter Overview of the CORBA Component Model, pages 557–571. Addison-Wesley, Boston, USA, 2001.