A Transparent Performance and Failure Management Service for CORBA based Applications

Carlo Emmanoel Tolla de Oliveira[1], Renato Fiche Junior[1]

[1] Núcleo de Computação Eletrônica – NCE,
Universidade Federal do Rio de Janeiro – UFRJ
Edifício do Centro de Ciências Matemáticas e da Natureza (CCMN), blocos C e E. Cidade Universitária – Ilha do Fundão – Rio de Janeiro, Brasil.

{carlo, renatofj}@ufrj.br
http://nisus.sourceforge.net

Abstract. In order to provide quality of service to enterprise applications, the need to find problems on their execution has calling organizations attention. As a result, applications management represents a fundamental aspect. However, it is hard to manage distributed applications due to their complexity and geographical decentralization. For these reasons, the aim of this paper is to present a management solution for any CORBA distributed application built with Java language. The idea consists of a service with components that work together in order to provide a robust management. The proposed solution allows a centralized, transparent and detailed performance and failure management. It is capable to find behavioral problems in specific parts of such applications. A prototypical implementation was used to experiment the solution. The results will be analyzed on this paper.

1 Introduction

Understanding the distributed applications behavior is one of the most challenging problems facing today's software engineer. Distributed applications require a completely different level of monitoring than traditional single-process applications. This is due to remote communication complexity and to their geographical decentralization. In [1], the author does a brief review of the most common problems that can be found during distributed applications execution: performance bottlenecks, network resource limitations, network failures, race conditions, deadlocks, design errors in control flow, timeout failures. That is, a range of complex problems can be found and as less time as possible is the reaction to such problems the application availability will be greater.

Besides software components of distributed applications are often localized in a considerable number of hosts. As a result of the distributed applications decentralization, data about their execution is often only available into log files dispersed for their environment. Thus performance and failure monitoring process becomes slow and tedious. A centralized solution is required in order to avoid this situation. Besides data

visualization should be friendly because developers shouldn't spend much time on performance and failure analysis.

In [2], the authors conclude that current techniques for performance monitoring of distributed applications are not sufficient to solve today's service management problems. They conclude that application instrumentation[1] is the only reasonable way to overcome such problems. However, due to the enormous efforts posed on the developer nowadays it is hardly used. Besides application code can become illegible and hard to be maintained when extra code is inserted to measure the application performance. For these reasons, application instrumentation should be automatic and. Thus a performance management solution shouldn't require manual work and should be easily integrated into existing distributed applications.

During software development for instance, the team shouldn't write pieces of code that are responsible to application response time measurement. Its productivity can be reduced because considerable effort is required to write them. While developers are in fact writing the application code, it is important they keep the focus as much time as possible on the most precious functionalities for the costumer. Thus functionalities tend to be working earlier. And as early as possible functionalities are working, the costumer feedback will be faster and the development team will have more time to do suggested changes.

Therefore, according to [3], traditional management platforms, like HP IT/Operations or Tivoli Management Environment (TME) are oriented more towards element management and often only provide a very limited type of application management. Currently tools to support understanding of how applications perform during operation are lacking [4]. Besides most management platforms treat the CORBA middleware as a black-box and thus give away valuable management information [3].

2 Purpose

The aim of this paper is to present an open performance / failure management solution for CORBA-based distributed applications – Nisus.

Nisus provides full transparence to developers, i.e., they don't need to write code to gather required information to application performance measurement. It is achieved through the CORBA feature called *Portable Interceptors.*

Nisus uses macroscopic and microscopic approaches combination. Through a Web console, it allows a detailed and centralized performance and failure analysis. Steps performed by each *transaction* can be viewed at the sequence they occur with *Performance Sequence Diagrams*. In the context of this paper, a transaction can be translated into "a call from a client to a server application". *Macroscopic* performance and failure analysis can also be done through graphical reports.

The console also shows the current load and running processes on servers and clients hosts. Nisus uses SNMP to gather client and server hosts' information such as CPU, memory usage, physical memory, and list of running processes.

---

[1] Application instrumentation means insertion of specialized management code directly into the application's code.

The remainder of this paper is organized as follows: Section 3 summarizes CORBA and Portable Interceptors. Section 4 presents a brief review of SNMP. Section 5 shows the proposed service by describing its components and how they work. Section 6 presents the results of the presented solution. Section 7 presents related work. Section 8 presents the future work and concludes the paper.

3 CORBA

CORBA is the acronym for Common Object Request Broker Architecture. It is an open, vendor-independent architecture and infrastructure created by OMG[2] that allows clients to invoke operations on distributed objects. CORBA achieves full interoperability in heterogeneous environments by providing location, programming language and platform transparency. Fig.1 shows components in the CORBA reference model [5].

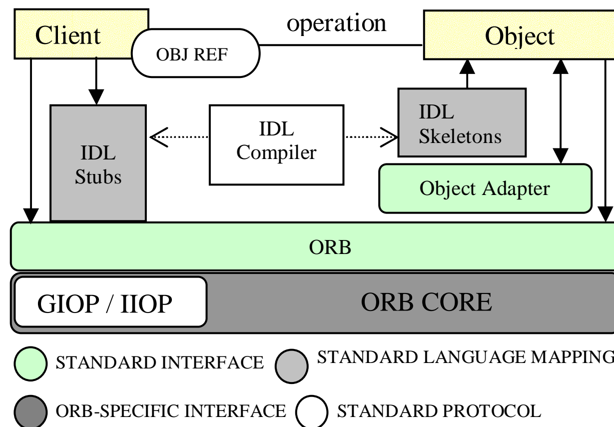


Fig. 1 – CORBA Reference Model

This paper only focuses on the CORBA's components relevant for the context of this paper. They are outlined bellow:

Client: A client is an application that obtains references to remote objects and invokes their operations.

Object: It is an instance of an IDL[3] interface. Each object represents an object reference. A server application can create several objects.

ORB Core: It is responsible for delivering all requests issued by clients and returning responses, if any. It is like a library linked into client and server applications. The nominated standard protocol IIOP[4] is used in the communication between client-side and server-side ORB.

---

[2] Object Management Group - http://www.omg.org
[3] Interface Definition Language
[4] Internet Inter-ORB Protocol

IDL Compiler: It transforms IDL definitions into stubs and skeletons that are generated automatically in an application programming language.

IDL Stubs: It represents original objects in a client application. When a client sends a request, stub transforms programming language representation of request into the IIOP representation.

IDL Skeletons: It represents original objects in a server application. When a request arrives, skeleton transforms request IIOP representation into the application programming language representation that is forwarded to the original object.

3.1 Portable Interceptors

The basic idea of Portable Interceptors is to allow developers to register interception code with the ORB that will be automatically executed in a *transparent* way for client and server applications when messages (e.g. transactions or replies) are exchanged between client-side and server-side ORB.

There are two types of request portable interceptors: *Client Request Interceptors* and *Server Request Interceptors*. The former are installed in client-side ORB and can intercept outgoing requests, incoming replies and exceptions. The latter, obviously, are installed in server-side ORB and can intercept incoming requests, outgoing replies and exceptions. Developers implement both interceptor types.

To write a client-side interceptor, the *ClientResquestInterceptor* local interface [6] shall be implemented. Client request interceptors are invoked when a client issues a request or when a reply or exception arrives. To write a server-side interceptor, the *ServerRequestInterceptor* local interface [6] shall be implemented. Server request interceptors are invoked when a server receives a request or when it sends a reply or exception. Fig. 2 shows such interception points.

Once installed, interceptors capture all requests and replies. Several instances can be registered with a single ORB, and all of them will be invoked when the event occurs. The order of execution of interceptors depends on the ORB implementation and cannot be either queried or modified.

In [7] the authors present two Portable Interceptors techniques: *Request Redirection* and *Piggybacking*. Portable Interceptors feature allows requests to be addressed to a different target from the one defined by ORB. Piggybacking mechanism can be used to transmit additional information onto messages (e.g. request or reply). It is fully transparent to client and server applications and improves communication between client-side and server-side interceptors.

Portable Interceptors can be used for several purposes. In according to [8], Portable Interceptors can provide client side enhancement like caching, load balancing, flow control and fault-tolerant. Load balancing mechanisms can be implemented through *Request Redirection* technique. In [9] the authors also present how Portable Interceptors can enhance CORBA applications. Protocol adaptation can be implemented in order to enable CORBA applications to communicate using protocols other than IIOP. Besides, CORBA does not currently include components for profiling and monitoring [9]. Profiling interceptors can be implemented in order to trace an application and its messages transparently, without the need to embed any special code into the application or the ORB. Others features like scheduling and reliability mechanisms can also

be implemented to override the ORBs thread policies and to detect and recover faults respectively.

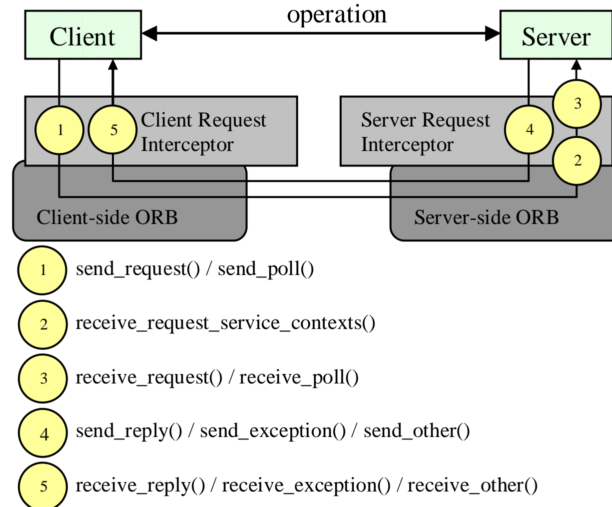


Fig. 2 – Client and Server Interception Points

4 SNMP

SNMP is the acronym for Simple Network Management Protocol. It was pronounced by the IAB[5] in 1988. It is an open standard designed originally to manage TCP/IP based networks. SNMP is implemented as a client-server model where the management application becomes the *client* and all the managed network elements become the *servers*. The network elements may be gateways, hosts, terminal servers, printers, hubs, bridges, etc. However, this paper only focuses in hosts CPU and memory usage monitoring.

SNMP is constituted by the following components: Management Protocol, Management Information Base (MIB) and Structure of the Management Information (SMI). MIB is a collection of all the variables that a client can manage such as hosts CPU and memory usage. SMI is a standard which the format of the MIB variables, i.e., identifier, data types, length, etc. The management protocol is very simple and defines basically four operations:

- Get: which is used to retrieve a specific MIB variable value from the managed network element
- Get-next: which is used to retrieve the next MIB variable value
- Set: which is used to set the value of an MIB variable

[5] Internet Activities Board

- Trap: which is initiated by the managed network element to report events

In [10], the authors present vantages of SNMP such as simplicity, extensibility, interoperability. Besides SNMP allows a centralized management and requires minimal resources. According to authors the main disadvantage is the proprietary MIBs. The SNMP implementations might have proprietary MIBs and for this reason interoperability might become difficult.

5 Solution

This section will show the components of the proposed management architecture and how they work to manage distributed applications performance and failure. Fig. 3 shows these components and where they can be localized.

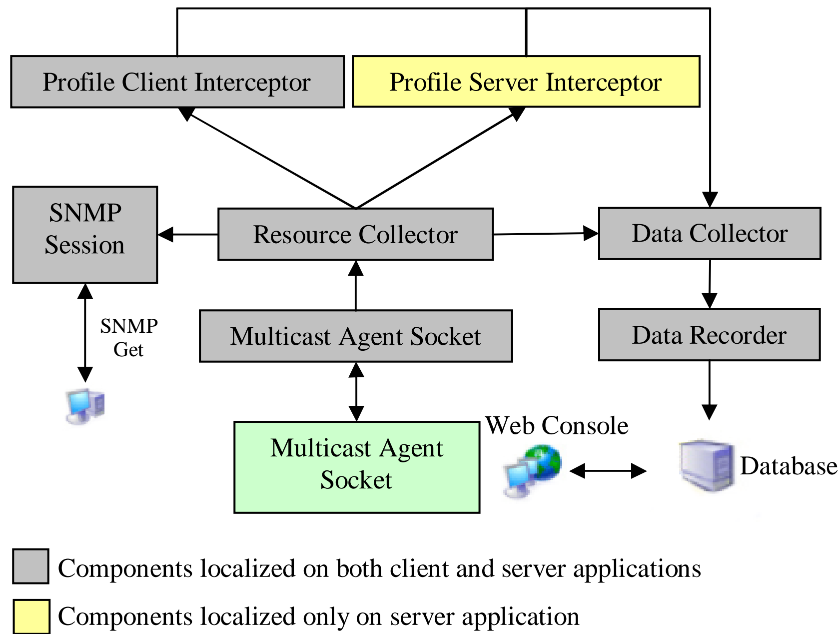


Fig. 3 – Nisus Components

These components are automatically installed when client-side and server-side ORBs are initialized. *Profile Client Interceptor* and *Profile Server Interceptor* are *Request Portable Interceptors* (see section 3.1). *Profile Client Interceptor* component is also installed on the server application in order to trace the requests issued within a transaction.

SNMP Session component is responsible to provide SNMP data. It is the client of the SNMP client-server model (see section 4). This component uses the AdventNet[6] SNMP implementation to gather information such as memory and CPU usage from the host where it is installed, i.e., client or server host. It encapsulates the API programming details, i.e., it is a *Wrapper* [11] design pattern implementation.

Client and server applications and Nisus Web console can exchange information with each other through *Multicast Agent Socket* component. Thus any Nisus web console instance can ask for status information and show it to the performance administrator (see section 5.2). Nisus Web Console is visualized through any browser and allows both macroscopic and microscopic analysis in a centralized manner.

5.1 Resource Collector

Resource Collector allows that resource information about clients and servers is recorded into the database. It can be further analyzed so that the application resource use is understood.

It gets SNMP data by calling *SNMP Session* in each minute. If this component is installed on client application, it also gets the number of transactions issued but not yet finished from *Profile Client Interceptor*. If it is installed on server application, it gets the number of transactions received but not yet finished from *Profile Server Interceptor*. After this, it sends the collected data to your *Data Collector* component reference. *Data Collector* buffers collected data and calls *Data Recorder* after each *x* seconds which can be configured (Fig. 4.a). That is, *Data Recorder* can submit SQL commands in *batch* to the database in order to avoid performance impact. A threshold can also be configured in order to avoid that excessive SQL commands are submitted at once (Fig. 4.b). If there is no need to analyze client or server hosts resource information, this component can be disabled (Fig. 4.c).

```
<client>
  …                                   (c)
  <resource-collector enabled="false">
    <recorder>
      <interval>240</interval> (a)
      <threshold>200</threshold> (b)
    </recorder>
  </resource-collector>
</client>
```

```
<server>
  …                                   (c)
  <resource-collector enabled="false">
    <recorder>
      <interval>120</interval> (a)
      <threshold>200</threshold> (b)
    </recorder>
  </resource-collector>
</server>
```

Fig. 4 – Nisus XML configuration file content for client and server applications *Resource Collector* configuration

[6] http://www.adventnet.com

5.1 Performance Sequence Diagram

Nisus web console can get transactions information from the database and translates it into a *Performance Sequence Diagram* which is visualized as java applet (Fig. 5). Performance Sequence Diagram allows a microscopic performance management. Thus performance problems can be easily identified in specific parts of CORBA application.

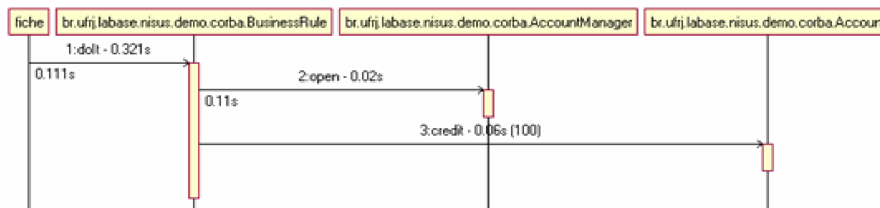


Fig. 5 – *Performance Sequence Diagram* example

Fig. 5 shows a *doIt* transaction issued from the client *fiche*. It taken 0.111 seconds until the request arrives the server. The first request issued within *doIt* transaction context is *open* which taken 0.02 seconds to be executed. After, it taken 0.11 seconds until *credit* request call. Nisus is able to identify simple loops when a transaction performance sequence diagram is built. So *credit* request was probably called 100 times within a loop as seen in Fig. 5.

When a client application issues a transaction request like *doIt*, the following steps are performed:

- *Profile Client Interceptor* is called and it generates a 32-bytes unique identifier which is sent to the *Profile Server Interceptor* through *Piggybacking* technique (see section 3.1).
- When a transaction arrives to *Profile Server Interceptor*, a *java.util.Stack* object is associated to the transaction thread (current thread). At this moment, the transaction identifier is pushed in the *Stack* object.
  If any CORBA request, i.e., sub-transaction is issued within the transaction context, *Profile Client Interceptor* is called. So it gets the top of the *Stack* object associated to the transaction thread (current thread) and associates it as the parent of the request. After this, it pushes the request identifier in the *Stack*. After the sub-transaction execution, its identifier is popped from the *Stack* and the *send_request* and *receive_reply* (or *receive_exception*) timestamps are passed to the *Data Collector*. *Data Collector* performs the same steps described in section 5.1. This process is done until the transaction execution end.
- When a transaction execution ends, *Profile Server Interceptor* is called and transaction arrival (*receive_request*) and processing end (*send_reply*) timestamps are passed to *Data Collector*. *Profile Server Interceptor* isn't called when server objects issue requests. It is only responsible for transactions network leaving time and their execution time measurement.
- After this, the *Profile Client Interceptor* localized on the client application is called. The *send_request* and *receive_reply* (or *receive_exception*) timestamps

are passed to the *Data Collector*. This *Profile Client Interceptor* instance is only responsible for transactions response time and their network return time measurement.

It is possible to enable/disable profiling interceptors through Nisus XML configuration file (Fig. 6.a). For example if there is no need to know the sequence of requests performed within a transaction context, the *Profile Client Interceptor* localized on the server application can be disabled. If there is no need to measure the transactions network trip time and their execution time, the *Profile Server Interceptor* can also be disabled. Thus, Nisus allows that only transactions response time is measured. In this case, only *Profile Client Interceptor* localized on the client application should be enabled.

```
<client>                                (a)
  <client-interceptor enabled=”true”>
    <recorder>
      <interval>40</interval>
      <threshold>1000</threshold>
    </recorder>
  </client-interceptor>
  …
</client>
```

```
<server>                                (a)
  <client-interceptor enabled="true">
    <recorder>
      <interval>40</interval>
      <threshold>1000</threshold>
    </recorder>
  </client-interceptor>                 (a)
  <server-interceptor enabled="true">
    <recorder>
      <interval>60</interval>
      <threshold>1000</threshold>
    </recorder>
  </server-interceptor>
  …
</server>
```

Fig. 6 – Nisus XML configuration file content for client and server profile interceptors configuration

It is also possible to configure filters in the Nisus XML configuration file in order to avoid that certain transactions and requests are intercepted by Nisus profile interceptors. These filters are configured with regular expressions.

5.2 On-line Resource and Load Monitoring

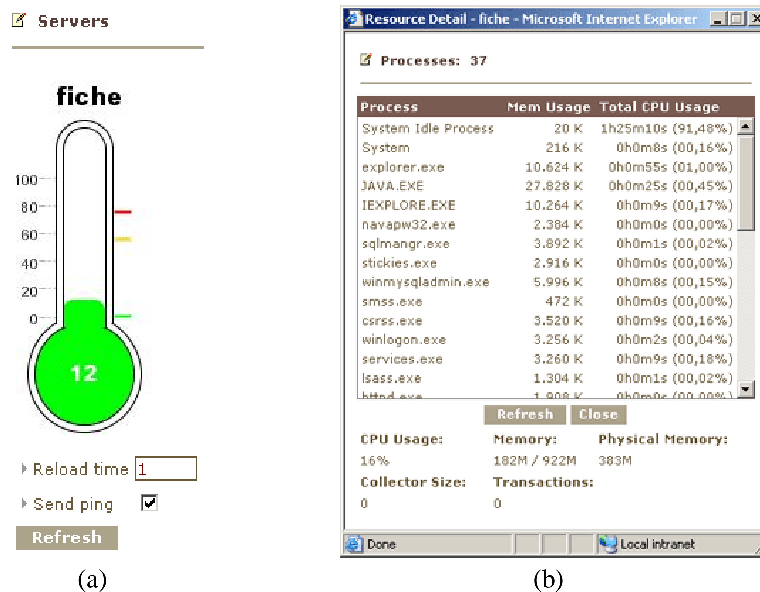
This feature was idealized on the application called *Broker* which was used in the old on-line disciplines registration system of UFRJ. It was used for years in order to show the current number of clients and the latency experimented for each web server. Round robin algorithm was also used in order to balance the number of clients among the web servers. In *Broker* the load is only based on the number of clients.

Nisus Web console allows on-line load and resource monitoring. Both client and server applications can be monitored. It is useful in 4-tier applications where for example web servers (clients) are in the second tier and CORBA applications (servers) are in the third tier. It is important to know how many transactions are in processing

state in each web server and CORBA application and whether they are running properly or not. The main difference between Nisus and *Broker* is that Nisus supports remote CPU and memory monitoring.

Through *Multicast Agent Socket*, Nisus Web console can send *Ping* objects to the multicast group. When a client and server applications receive a *Ping* object, they answer with a *ResourceData* object which contains detailed resource information.

The client or server load is represented by a single value and a metaphor with *temperature* was used to better explicit the information meaning (Fig. 7.a). Through a mouse click on the thermometer picture, a pop-up window is opened where information details such as current running processes, total number of transactions in processing state, total memory and CPU usage are showed (Fig. 7.b).



(a) (b)

Fig. 7 – Server *fiche* temperature (a), Running processes list, CPU and memory usage, number of transaction in processing state (b).

The load is a number between 0 and 100 based on such detailed data and is calculated through the formula (1).

$$\text{Load} = \textit{Temperature} = (\text{Memory usage \%} + \text{CPU usage \%} + (\textit{Formula 2})) / 3 \qquad \textbf{(1)}$$

$$\text{Transaction Load} = (n / t)*100\ \% \qquad \textbf{(2)}$$

Where n is the number of transactions in execution state and t is the maximum number of transactions which has experimented by the application.

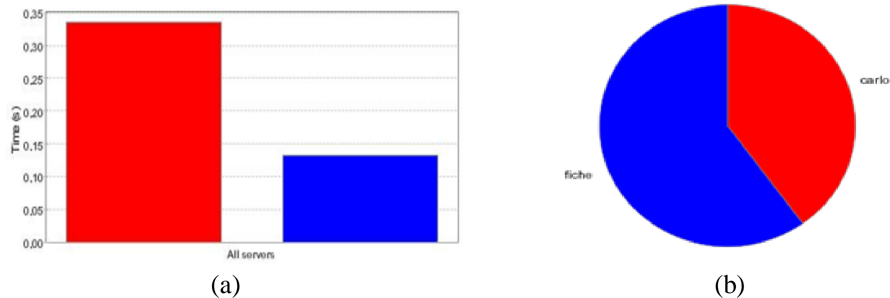
The t parameter in (2) may be different from an application to another. Initially, this parameter value may be sufficiently large. Ideally, the parameter value should be

the larger number of transactions in execution state observed. Since Nisus monitors that number (see Section 5.1), this parameter can be better configured according to time.

5.3 Reports

Nisus allows a macroscopic performance and failure management. It provides reports so that management data recorded by the presented solution components is analyzed and the application behavioral is understood. Thus it is possible to easily identify performance problems and application faults.

Fig. 8 shows two reports provided by Nisus through which performance and behavioral problems can be easily found. Fig. 8.a shows an example where there is a significant difference between the transactions *response* time average (red bar) and the transactions *execution* time average (blue bar). It can be caused by network traffic for example. Thus transactions parameters can be reviewed. This report can be limited to a certain object, operation, date, client or server. It can also be grouped by server. Fig. 8.b shows the percentage of transactions received by server. Thus, it is easily to know whether the application is distributing transactions equally among all servers or not.



(a) (b)

Fig. 8 – Nisus Reports

Nisus also provides reports such as number of faults per transaction, CPU and memory usage and the number of transactions in execution state according to time. Thus they can be used to understand the application resource consume and to know when the application experiments high load and which parts of the application have more faults.

6 Results

Nisus was applied to a sample bank CORBA application. The experiment consists in two workstations interconnected by a 100Mbit Ethernet LAN: (1) equipped with 750Mhz Athlon processor, 128Mbyte of RAM. (2) equipped with 750Mhz Pentium III processor, 384Mbyte of RAM. The workstations run Microsoft Windows XP Pro-

fessional as operational system. VisiBroker 5.0 and OpenORB 1.2 ORBs were used for Portable Interceptors implementation performance comparison. MySQL 4.0.13 for Windows was used to record the collected data.

The experiment client application was installed on the workstation (1) and launches 100 threads. Each one sends 10 requests to a *BusinessRule* CORBA object *doIt* method. This object was localized on the workstation (2). Fig. 9 shows a transaction *doIt* Performance Sequence Diagram.

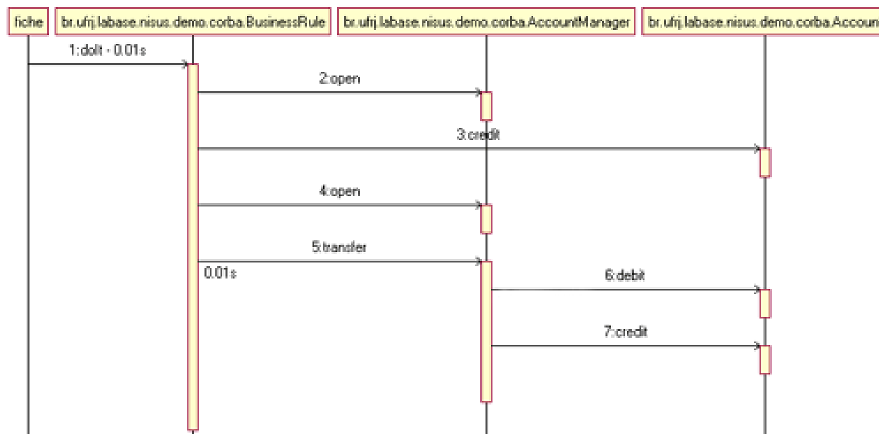


Fig. 9 – *doIt* transaction Performance Sequence Diagram

In order to measure the Nisus and Portable Interceptors overhead, end-to-end latency was measured for the following cases:

- Case 1 – Without Nisus profile interceptors
- Case 2 – Only with *Client Profile Interceptor* installed on the client application
- Case 3 – With *Client Profile Interceptor* installed on the client application and *Server Profile Interceptor* (installed on the server application)
- Case 4 – All Nisus profile interceptors installed
- Case 5 – With an empty *Client Request Interceptor* installed on both client and server applications and an empty *Server Request Interceptor* installed on the server application
- Case 6 – With an empty *Client Request Interceptor* installed on the client application and an empty *Server Request Interceptor* installed on the server application

The experiment was executed 5 times for each case generating a total of 5000 transactions for each case. The record interval was set to 40 seconds for *Client Profile Interceptor* and 60 seconds to *Server Profile Interceptor*. The threshold was set to 1000. Since the record interval is sufficiently large, the collected data is only recorded after the transactions completion. Thus the *Data Collector* wasn't executed during the experiment measurements and will not be considered in the results.

Fig. 10 shows the results for VisiBroker. When only the response time is measured (Case 2), the overhead is practically 0%. Thus we concluded that an empty *CORBA Server Request Interceptor* has an overhead about 3.1% (Case 6 - Case 2) and the *Client Profile Interceptor* overhead is about 0% when it is installed on the client application. When the response, execution and transactions trip time are measured (Case 3), the overhead is about 10.1%. That is, 7% (Case 3 - Case 6) is caused by *Server Profile Interceptor* Nisus component. When the *Client Profile Interceptor* is installed on the server application (Case 4), the overhead is about 32.6%. An empty *CORBA Client Request Interceptor* installed on the server application has an overhead about 6.7% (Case 5 - Case 6). Thus we conclude that *Client Profile Interceptor* installed on the server application has an overhead about 15.8% (32.6% - 6.7% - 10.1%). In the case 4, more than 100 transactions presented response time above 5 seconds. Understanding the reasons for this situation will help in the Nisus performance improvement.

	Average (s)	Median (s)	Minimum (s)	Maximum (s)	Std. Dev. (s)	Over-head
Case 1	1.080	1.001	0.220	2.083	0.464	0%
Case 2	1.064	0.962	0.000	2.224	0.468	0%
Case 3	1.201	1.192	0.000	2.233	0.481	10.1%
Case 4	1.604	1.412	0.240	7.130	0.932	32.6%
Case 5	1.198	1.111	0.280	2.113	0.412	9.8%
Case 6	1.115	1.052	0.050	2.083	0.441	3.1%

Fig. 10 – Experiment results for VisiBroker

In the experiment for OpenORB, we observed that the *Profile Client Interceptor* component wasn't called when it was installed on the server application. For OpenORB implementation, we conclude that *Client Request Interceptors* are only called if the target object is localized on another ORB instance. For this reason, the results for Cases 4 and 5 weren't presented.

Fig. 11 shows the results for the OpenORB. When only the response time is measured (Case 2), the overhead is about 3.4% against 0% from VisiBroker. Thus we concluded that *Server Profile Interceptor* has an overhead about 2.4% (Case 3 - Case 2) against 10.1% from VisiBroker. This overhead already includes the empty *CORBA Server Request Interceptor* overhead. Understanding the reasons for this difference will also help in the Nisus performance improvement for VisiBroker.

	Average (s)	Median (s)	Minimum (s)	Maximum (s)	Std. Dev. (s)	Over-head
Case 1	1,102	0,671	0,350	5,929	1,428	0%
Case 2	1,141	0,681	0,240	5,999	1,438	3.4%
Case 3	1,169	0,721	0,291	6,019	1,437	5,7%
Case 4	-	-	-	-	-	-
Case 5	-	-	-	-	-	-
Case 6	1,129	0,671	0,090	5,969	1,433	2.4%

Fig. 11 – Experiment results for OpenORB

7 Related Work

For CORBA applications, there are similar works. In [3], a generic instrumentation approach is also presented. It is based on Portable Interceptors and ARM[7]. Through ARM, nested unit of works can be measured. However, the results concluded that the solution presented high overhead. For example the ARM library presented overhead about 65%. This work doesn't concern about how performance data is visualized.

In [4], the authors present an approach to allow a developer to understand the runtime behavior of a distributed application. A visualization tool is used to study the statistics. However this work doesn't provide resource monitoring and on-line performance and failure monitoring.

In [12], the authors present a framework for collecting performance statistics. As it is also based on Portable Interceptors, applications source code modification isn't required. It measures CPU and memory usage for each transaction for use in capacity planning. However, performance information is only stored into log files. It provides a visualization tool but data can be only analyzed when application is running. Besides it only uses macroscopic approach.

CorbaTrace[8] project also uses Portable Interceptors so that communication among CORBA objects are recorded into XML files. For better visualization, theses files can be transformed into XMI files which can be viewed as sequence diagrams in any UML tool. However, they don't show information about application performance. Thus this project can only be used for automatic documentation purposes.

There are also solutions for EJB[9] distributed applications. In [13], the authors present a framework for performance monitoring of component oriented distributed applications based on the EJB specification. It is also generic and non-intrusive. Performance information is sent to a messaging queue through a JMS implementation. A graphical console reads the messages from the queue and displays performance information. However, it is only able to provide real-time performance analysis, limited information and few graphical views as well. In [14], the authors present a generic approach for EJB applications performance evaluation using JMX[10], but their work doesn't present concern about performance data visualization.

8 Conclusions and Future Work

The presented solution – Nisus – combines macroscopic and microscopic performance and failure management. Besides it allows that application resource consume is identified. It is also able to show the current load on clients and servers and allows that the developers don't spend much time during management analysis. It is achieved because Nisus has a friendly web console interface. Through this console, Nisus also provides a centralized and remote performance and failure management.

---

[7] Application Response Measurement
[8] http://corbatrace.tuxfamily.org
[9] Enterprise Java Beans
[10] Java Management Extensions

Nisus is fully transparent to developers. It is not require that extra code is inserted into the applications code to measure their performance and monitor the resources used by them.

We concluded that Nisus overhead can vary from an ORB to another. We also observed that CORBA Portable Interceptors isn't fully portable. This paper showed that there are differences between the two ORBs Portable Interceptor implementations – VisiBroker 5.1 and OpenORB 1.2.

The main future work is to increase the *Client Profile Interceptor* component performance when is installed on the server application. Without this component, it is not possible to identify the sequence of requests performed within the transactions.

Nisus is being integrated into the academic management system of UFRJ – SiGA. It has more than 60.000 users. The aim of SiGA is to control all academic processes of UFRJ. In order to show that the presented solution can be easily installed into a production environment, developers that aren't working in Nisus implementation are doing the integration into SiGA.

A *temperature* (see section 5.2) can be used for load balancing purposes. As client and server applications are joined in a multicast group, they can exchange load information with each other and a load balancing mechanism can be implemented by using the Portable Interceptors redirection technique (see section 3.1). A tolerant fault infrastructure such as MEAD[11][15] could also be incorporated into Nisus in order to increase CORBA distributed applications dependability.

References

1. Scallan, T.: Monitoring and Diagnostics of CORBA Systems. Java Developers's Journal, Vol. 5. (2000) 138-144.
2. Hauck, R, Radisic, I.: Service Oriented Application Management – Do current techniques meet the requirements? 3rd IFIP International Working Conference of Distributed Applications and Interoperable Systems. (2001).
3. Debusmann, M., Schmid, M., Kroeger, R., Wiesbaden, F.: Measuring End-to-End Performance of CORBA Applications using a Generic Instrumentation Approach. IEEE 7th International Symposium on Computers and Communications. Taormina-Giardini Naxos, Italy. (2002) 181-187.
4. Moe, J., Carr, D. A.: Understanding Distributed Systems via Execution Trace Data. IEEE 9th International Workshop on Program Comprehension. Toronto, Canada. (2001) 60-67.
5. CORBA (The Common Object Request Broker Architecture) Specification, 2003.
6. CORBA Portable Interceptors Specification, 2003.
7. Marchetti, C., Verde, L., Baldoni, R.: CORBA Request Portable Interceptors: A Performance Analysis, 3nd International Symposium on Distributed Objects and Applications. (2001) 208-227.
8. Friedman, R., Hadad, E.: Client-side Enhancements using Portable Interceptors. IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. Rome, Italy. (2001) 179-185.
9. Narasimhan, P., Moser, L. E., Melliar-Smith, P. M.: Using Interceptors to Enhance CORBA. IEEE Computer, Vol .32. (1999) 62-68.

[11] http://www.ece.cmu.edu/~mead

10. K. Amirthalingam, R. J. Moorhead.: SNMP-an overview of its merits and demerits. IEEE Southeastern Symposium on System Theory. Starkville, USA. (1995). 180-183.
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software (1999).
12. Sridharan, B., Dasarathy, B., Mathur, A.P.: On building non-intrusive performance instrumentation blocks for CORBA-based distributed systems. IEEE Computer Performance and Dependability Symposium. (2000) 139 -143.
13. Mos, A., Murphy, J.: Performance Monitoring of Java Component-Oriented Distributed Applications IEEE International Conference on Software, Telecommunication and Computer Networks. Croatia, Italy (2001).
14. Debusmann, M.; Schmid, M.; Kroeger, R.: Generic performance instrumentation of EJB applications for service-level management. IEEE/IFIP Network Operations and Management Symposium. (2002) 19 -32.
15. Narasimhan, P..: MEAD: Real-time Fault-Tolerant Support for Middleware. OMG Real-Time Special Interest Group Meeting on Fault Tolerance. Orlando, USA. (2003).