Distributed Object-Oriented Applications Supervision

N. Cottin, J. Gaber, O. Baala, M. Wack
{Nathanael.Cottin, Jaafar.Gaber, Oumaya.Baala, Maxime.Wack}@utbm.fr
University of Technology of Belfort-Montbeliard
90010 Belfort, France

Abstract

Distributed object-oriented computing allows efficient use of the Network Of Workstations (NOW) paradigm. However, the underlying middlewares used to develop and deploy such applications do not provide developers with any standard supervision mechanism so that they know exactly what happens during their applications execution.
This paper analyzes distributed CORBA and JAVA-based applications to point out functional and management supervision information which has to be gathered from the objects. Developers will use this information to improve the Quality of Service (QoS) of their distributed object-oriented applications (DOA).

Keywords : Distributed computing, supervision, QoS, JAVA, CORBA.

1 Introduction

We consider distributed applications from a supervision perspective. Supervision is based on observation whereas management uses observed information to perform modifications on the supervised entities. Thus, managing distributed object-oriented applications (DOA) requires supervision mechanisms. Management systems rely on ad-hoc supervision indicators to take QoS improvement decisions. Management tools developement is a complex process. Developers tasks can be simplified by using a supervision API. In this paper, we present a supervision API and propose supervision indicators relevant to a management system. We focus on CORBA and JAVA-based applications [1] to show how this API could be implemented.
The rest of the paper is organized as follows : next section provides an overview of the work related to the distributed applications QoS improvement issue. Then we present the background material necessary to seize our study. This includes distributed applications representation and supervision implementation. These two sections lead to introduce a supervision API inherited by distributed objects as well as supervision indicators based on this API. We finally conclude by presenting our work in progress following our study.

2 Related work

Some projects have proposed ways to improve distributed CORBA-based applications QoS in terms of response time and overhead. Among them, QUARTZ, LSS and DOMS are representative of this research area. However, QUARTZ and LSS are dedicated to special-purpose applications such as media and service-based applications whereas DOMS is a general-purpose QoS improvement system. Another way of supervising distributed applications is to directly instrument the ORB [2]. However, this solution is ORB-dependant and not portable.

2.1 QUARTZ

Quartz [3] is a generic QoS description environment integrated with CORBA developped at Trinity College Dublin. It is used to guarantee that CORBA-based distributed applications meet specified QoS criteria for control and transfert of streaming media.
Quartz interprets QoS parameters required by applications to allocate corresponding resources at system-level. It uses application and system-level filters to translate QoS constraints and allocate resources according to various protocols such as RSVP, TCP/IP and ATM. A C++ prototype using Iona Orbix has been developped. It uses Windows NT real time capabilities as well as the audio and video streaming mechanism standardized by the OMG [4].

2.2 LSS

The IT Research Center of Montreal has designed a service facility called Load Sharing Service (LSS) [5]. This distributed CORBA-based system is used for managing objects calls when multiple objects provide the same service type.

This system uses a service-request point of view to perform load sharing strategies. Distributed objects register to the CORBA trading service [6] and give the services they are able to afford. These objects are called Servers because they are called by Client objects to perform tasks. The load sharing algorithm uses Servers waiting queue lengths to send Client requests to the less loaded Servers.

2.3 DOMS

Distributed Objects Management System (DOMS) [7] is a project led by UTBM. It addresses the problems of CORBA-based distributed objects supervision, applications management and QoS improvement. Functional and management information from objects and computers is gathered by agents running on each computer over the network. This information is then sent to a supervision manager which takes objects migration decisions depending on objects migration ability.

3 Notations

A distributed object-oriented application is a set of distributed objects which interact with each other locally or over a network to perform specific tasks. It can be considered as a dynamic directed-graph $G = (O, L)$ where O is the set of distributed objects composing the application and L the set of edges representing links between these objects :

- Distributed objects are assimilated to nodes. At our supervision level, objects are considered as entities which may receive and generate calls
- Edges represent dynamic links between objects. Each edge is oriented from a source to a destination object. It is weighted by the number of calls which can be assimilated to the degree of correlation between these two distributed objects.

4 Supervision implementation

Distributed applications supervision encounters two main issues :

- Find a way to *supervise* distributed objects. This must be as less intrusive as possible to be easily integrated into existing applications
- Allow *observed* objects *traceability* using a reference-independant *identification* process.

4.1 Supervision API utilization

The main problem encountered when designing a JAVA-based management system is to integrate supervision and management mechanisms into the applications.
Adding supervision functionalities to distributed objects can be done by making distributed objects inherit supervision methods from a supervision API [8].
However, CORBA and JAVA-based objects interfaces already extend the “org.omg.CORBA.Object” class. Instead of directly inherit from this class, distributed objects (DO) should inherit from a medium supervision class *ObservableDO* as described on figure 1.

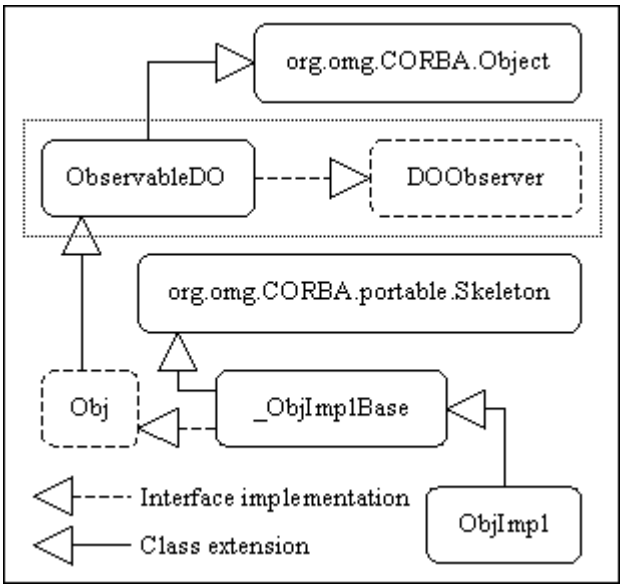

Figure 1. Supervision API inheritance classes.

Every distributed object skeleton would then inherit from an *ObservableDO* class implementing this API to allow objects supervision :

```
import org.omg.*;

public class ObservableDO
  extends CORBA.Object
  implements DOObserver {
  ...
}

public interface Obj
  extends ObservableDO {
  ...
}
```

```
abstract public class _ObjImplBase
  extends CORBA.portable.Skeleton
  implements Obj {
  ...
}

public class ObjImpl
  extends _ObjImplBase {
  ...
}
```

This instrumentation may be processed during object's IDL compilation. Management agents (i.e. daemons) running as a CORBA service on each computer will therefore interact with the supervised objects by calling their inherited supervision methods. The overload due to this supervision mechanism depends on agents activity.
However, supervision systems need to be able to uniquely identify each *observable* object supervised to be able to exploit supervision information gathered by every daemon.

4.2 Distributed objects identification and traceability

Implementing objects supervision includes objects identification. The identification process must be reference-independant to allow traceability.
Although using objects references is sufficient to uniquely identify a given object, it becomes obsolete when this object migrates from its local host to a distant host. Therefore, its IOR reference has changed.
This identification should proceed from the underlying middleware to make sure identifiers are not duplicated, considering that many objects can instanciate the same class.

5 Supervision API

5.1 Definition

We define a distributed objects interface called *DOObserver* to implement. This abstract class is declared using the interface description language designed by the OMG [9]:

```
typedef sequence<string> set;

interface DOObserver {
  attribute string ID;
  string getHostName();
  long long getCPU();
  long long getMEM();
  set getCalledObjectsID();
  long getCallsNumber(in string ID);
  long getWaitingQueueLength();
  long long getCallCompletionTime();
  long long getAverageExecutionTime();
  void objectCall(in string ID);
}
```

5.2 Description

This supervision interface implemented within a given distributed object i is described as follows:

- The *ID* attribute represents i's *identifier* mentioned in the previous section
- *getHostName()* returns the i's local host identifier. This identifier can either be an IP address or a DNS name
- *getCPU()* and *getMEM()* methods express the i's activity in terms of processor and memory consumption
- *getCalledObjectsID()* is udes to retrieve the set of objects called by i since its creation date
- *getCallsNumber(ID)* returns the number of calls generated by i to the object identified by *ID*
- *getWaitingQueueLength()* return the number of objects waiting for a task handled by i and not currently completed. This method is used by load sharing management systems
- *getCallCompletionTime()* returns the amount of time necessary to receive the last invocation result given by the last called object
- *getAverageExecutionTime()* returns the average amount of time necessary to locally handle objects calls
- *objectCall(ID)* is executed each time i calls another distributed object. It updates the G graph edges weights. This method is complementary to *getCallsNumber()*.

This supervision API may be extended to allow objects management by adding self-migration capabilities [10] [11] and objects notifications.

6 Supervision indicators

Based on the objects relationships decribed by the application G graph and our supervision API, we define supervision indicators related to distributed objects, objects classes and applications.

6.1 Object-level indicators

We define in this section indicators related to distributed objects. We consider each object independantly from other objects and from their environment.
We point out four object-level indicators :

- $ocor_{i,j}$ is the correlation degree between i and j objects. It corresponds to the *getCallsNumber()* method call
- $odiff_i$ is i's scatter level. The more objects are called, the higher $odiff_i$. This indicator also depends on objects calls number. It is calculated using the *getCalledObjectsID()* method
- $ocalls_i$ is the average number of calls initiated by i calculated with the *getCallsNumber()* method from the supervision API
- $orel_i$ represents i's functional reliability. The more objects belonging to different classes are called, the less reliable i and the lower $frel_i$. This indicator is obtanied by combining the *getCalledObjectsID()* and *getCallsNumber()* methods.

Usually, $ocor_{i,j} \neq ocor_{j,i}$:

- $ocor_{i,j} = 0$ and $ocor_{j,i} > 0$ means that j should migrate to i's local host. This relationship is *cooperation unilateral*
- When $ocor_{i,j} > 0$, $ocor_{j,i} > 0$ and $ocor_{i,j} - ocor_{j,i} > \varepsilon$, i and j are *cooperative bilateral*. The migrated object has the highest object correlation indicator
- In case $ocor_{i,j} > 0$, $ocor_{j,i} > 0$ and $ocor_{i,j} - ocor_{j,i} \leq \varepsilon$, i and j are *cooperative symetrical*. Depending on i and j migration ability, i can either migrate to j's local host or j migrate to i's local host
- At last, j is *cooperative unilaterally excluded* from i's perspective when $ocor_{i,j} = 0$. i and j are *cooperative bilaterally excluded* when $ocor_{i,j} = ocor_{j,i} = 0$. In this case, i and j should migrate in order to be hosted by two different computers.

When used by a management system, these indicators express objects weights and reliability. Highly correlated objects will be migrated so that they perform as many local calls to each other as possible.

6.2 Class-level indicators

Class-level indicators caracterize objects classes relationships. They generalize object-level indicators to consider distributed objects at a higher level of abstraction :

- $ccor_{a,b}$ expresses the correlation degree between a and b classes. It represents the average correlation degree between all the objects belonging to class a and class b objects
- $cdiff_a$ is the diffusion degree of class a. It corresponds to the average diffusion level of each object instanciating this class
- $ccalls_a$ is the average number of calls of every distributed object of class a
- $crel_a$ stands for class a functional reliability. This indicator is based on class a objects $orel_i$ indicator.

Similarly to object-level relationships, we distinguish classes *cooperative exclusion*, *cooperative symetry*, *unilateral cooperation*, *bilateral cooperation*, *unilateral cooperation exclusion* and *bilateral cooperation exclusion*.

These indicators are useful when making objects activity predictions. They inform the management system on the possible interactions frequency between objects and allow to compare classes call frequency.

6.3 Application-level indicators

We previously introduced indicators based on object-level and class-level considerations. We now introduce objects environment to define application-level indicators. We particularly distinguish *local* and *distant* calls between objects. *Local* calls refer to invocations related to objects running on the same computer whereas *distant* calls occur between object hosted on diffrent computers. The call locality is established using the *getHostName()* method of the supervision API :

- $inside_i$ represents the number of local calls generated by a given object i
- $outside_i$ is the number of distant calls issued by i to other objects.

We then define an indicator to express the degree of membership of a given object i on its local host depending on i's local and distant calls :

$$membership_i = \frac{inside_i}{(inside_i + outside_i)} \times 100$$

This *membership* degree leads to define a similar indicator for a given computer c which belongs to the set of supervised computers C. The higher *membership* degree of local objects, the more excluded the computer to the rest of the system :

$$exclusion_c = \frac{\sum_{i \in c} membership_i}{Card(C)}$$

This *exclusion* indicator is used to validate management decisions taken by the management system in terms of distributed objects location optimization.

7 Conclusion

We introduced in this paper a supervision API which provides developers with supervision methods which allow to implement management algorithms. We used this API to define supervision indicators to value the QoS level of distributed CORBA and JAVA-based applications in terms of objects relationships.
Distributed applications QoS improvement is either achieved by reserving necessary resources or migrating objects. Our supervision API and indicators are useful in both cases :

- The supervision API includes objects processor and memory consumption for their activity to take resources reservation and objects migration decisions. Using a reference-independant object *identifier* allows to perform objects follow up continuously
- The supervision indicators are used to take migration and placement decisions when coupled with group reorganization based on strongly connected components within cooperative graphs [12].

We are currently working on DOMS [7], a CORBA-based distributed applications QoS improvement prototype implemented with OOC ORBacus for JAVA. It uses our supervision indicators to provide developers with a management API to integrate management algorithms into their supervised applications.

References

[1] J. Daniel, “Au cœur de CORBA avec Java”, Vuilbert, Paris, ISBN 2-7117-8659-5, 2000

[2] M. Wegdam, D.-J. Plas, A. van Halteren, B. Nieuwenhuis, “ORB Instrumentation for Management of CORBA”, Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’00), Las Vegas, NV, June 2000

[3] F. Siqueira, “Quartz : A QoS Architecture for Open Systems”, Ph.D. thesis, Trinity College, University of Dublin, December 1999

[4] Iona Technologies, Lucent Technologies and Siemens–Nixdorf, “Control and Management of Audio/Video Streams”, OMG/97-05-07, July 1997

[5] E. Badidi, R. K. Keller, P. G. Kropf, V. Van Dongen, “The Design of a trader-based CORBA Load Sharing Service” Proc. of the Twelfth International Conference on Parallel and Distributed Computing Systems (PDCS’99) Fort Lauderdale, FL, August 1999 (to appear)

[6] Object Management Group, “Trading Object Service Specification”, formal/2000-06-27, May 00

[7] N. Cottin, “DOMS : une architecture de suivi de la qualité de service dans les systèmes répartis objet”, Research report, Universtity of Technology of Belfort-Montbéliard, France, August 2000

[8] N. Cottin, O. Baala, J. Gaber, M. Wack, “Management and QoS in Distributed Systems”, Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’00), Vol. III, Las Vegas, NV, June 2000

[9] Object Management Group, “OMG IDL to Java Language Mapping Specification”, formal/99-07-53, June 1999

[10] T. Schneckenburger, “The Migration Pattern”, Component Users Conference, Munich, Germany, July 1997 (to appear)

[11] M. C. Pellegrini, “Reconfiguration d’applications réparties : application au bus logiciel CORBA”, Ph.D. thesis, Institut National Polytechnique de Grenoble, October 2000

[12] H. Adoud, E. Rondeau, T. Divoux, “Configuration Of Network Architectures For Co-operative Systems”, Proc. of the 26th Euromicro Conference (Euromicro Workshop On Multimedia and Telecommunications), Maastricht, the Netherlands, September 2000