Dynamic Adaption of Resource Aware Distributed Applications

Narkoy Batouma, Jean-Louis Sourrouille
INSA-Lyon, LIESP
F-69621, Villeurbanne, France
{narkoy.batouma, jean-louis.sourrouille}@insa-lyon.fr

Abstract

Dynamic adaptation has become an important issue when designing and developing distributed applications in order to manage their Quality of Service (QoS). This is especially challenging when distributed applications run in environments in which resources vary unpredictably over time. To deal with fluctuations in resource availability and inherent heterogeneity of distributed environments requires dynamic adaptation of each application. This trend motivates the design of resource-aware applications ensuring a given performance level by adapting their behavior to changing contexts. To tune the use of resources, adaptive systems include the necessary mechanisms to modify applications' behavior.

This paper presents a general distributed middleware for enabling behavior adaptation of distributed applications. The middleware combines application designer specification of alternative execution behaviors with information about the execution environment context for deciding when and how to adapt itself according to the available resources. A description language specifies applications' behavior and the description of their related resource use. The QoS management requires a common execution model for all applications.

Simulations of the QoS management of heterogeneous applications illustrate our proposal and show the benefits. In addition, we discuss lessons learned from our experience.

Keywords: *Behavior Adaptation, Adaptation Coordination, Application Model, QoS Management, Multi-Agent Systems.*

1. Introduction

In an open environment, distributed applications with time-dependent resource requirements face to context fluctuations such as resource availability, network connectivity. These unforeseen context can affect the Quality of Service (QoS) provided by applications. The need to take into account context fluctuations, and QoS requirements to provide end-to-end QoS support, becomes more and more insistent. This is especially challenging in open environments with no underlying operating and networking system support for QoS [1][2]. The difficulty lies in unpredictability in resource availability when many distributed applications share resources such as processor, network bandwidth or memory, and in the inherent heterogeneity of distributed environments.

To address these issues, designers use either a reservation approach that controls admission and reserves resources to execute the applications, or an adaptation approach that proposes mechanisms to tune application requirements according to the execution context. The adaptation approach has the advantage of better controlling the execution of applications according to the available resources. Adaptation mechanisms aim to make the most effective use of resources in order to provide an acceptable level of QoS and to adapt to changing conditions of systems and networks [3] [4][5][6][7][8].

Numerous works have been proposed for QoS management. They tackle the problem at different levels: at low level, by introducing guarantees in communication protocols; at intermediary level, by modifying the operating system resource management policy; at high level, by introducing a middleware to manage the QoS. Our works stand at the high level: applications hold the required knowledge to set their alternative behaviors. They provide a description of their behavior and resource requirements to achieve adaptation under the control of a middleware.

We built and evaluated centralized approaches for QoS management in earlier works [7] and [8]. The former uses scheduling mechanisms to manage QoS while the latter is based on a reinforcement learning technique. Besides, we experimented with a decentralized approach [9] based on borrowing mechanisms for resource management that aims to guarantee application execution. In this paper, we investigate a decentralized middleware to adapt applications' behavior. The middleware addresses the problem of how to change application behavior at runtime in order to adapt to variations in the execution environment, such as resource availability. We design a framework that is based on a general application model as the basis to support the construction of adaptive distributed applications and proposes several strategies for adaptation.

The paper is organized as follows: first, we present the overview of previous works (section II), the requirements and choices (section III). Then, we describe our approach to adapt applications (section IV) and show how the system is implemented in practice (section V). Finally, we give some results (section VI) and after discussions (section VII), we compare our approach to related works (section VIII) before concluding (section IX).

2. Previous Works

2.1. Vocabulary

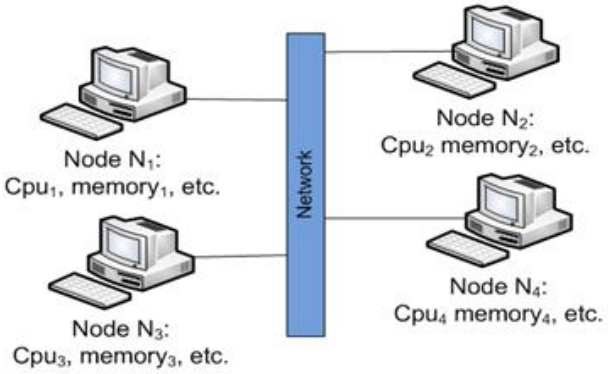

Figure 1. The Environment

In this paper we use some words or expressions which have the following meaning. A node ($Node_i$) is a separate piece of hardware that has at least one processor as in UML [10]. A resource is anything an application uses to execute. There are two types of resources: local resources (e.g., CPU, Memory) and shared resource (e.g., network bandwidth). A Local Manager (*LM*) is an entity that manages resources on a node. Our environment consists of a set of nodes connected via network (Fig. 1).

In this decentralized system the control is distributed between *LMs* that have only a partial knowledge of the system and must deal with communication explicitly to coordinate

their actions. A local application has one part on a node whereas a distributed application has parts on several nodes.

2.2. Previous Works

In decentralized cooperative systems using scheduling, autonomous *LMs* make decision based on their local view only. To improve the overall QoS of the system, the *LMs* must share information to coordinate their decisions i.e., the *LMs* must synchronize themselves so that they all observe the same current global system state. It would be possible for a *LM* to have information about the whole system by exchanging messages. However, it would be impractical due to the overhead costs, e.g., bandwidth cost. The aim of the previous work was to reduce the number of exchanged messages and to coordinate the planning of *LMs* in order to allow making sound decisions. In the sequel we describe briefly our previous works that are detailed in [9]. The previous approach presents a distributed middleware to manage resources in a decentralized manner. It is based on the use of approximate scheduling and a resource-borrowing schema to manage QoS. It aims to guarantee that as many applications as possible meet their deadline, to decrease the message exchange costs while increasing the use of resources. According to this approach, the time is subdivided into periods of length T and at the beginning of a period, each *LM* keeps a part of its resources (α% of T) and loans the remainder (β% of T) to other nodes. Shared resources such as bandwidth are managed in the same way (each node reserves σ% of T). Fig. 2 gives a representation example for three nodes $Node_1$, $Node_2$, $Node_3$ with respectively local resources Cpu_1, Cpu_2, Cpu_3 and a network shared resource Net. Here α=50, β=25 and σ=100/3, 3 is the number of nodes.

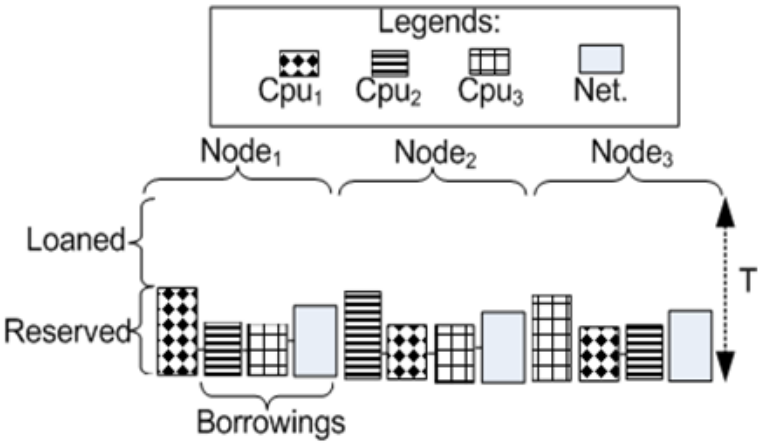

Figure 2. Example of Policy

The Fig. 2, shows the resources that a node distributes to other nodes at the beginning of a period can be viewed: (i) from a node that distributes resource side, it is called *loaned resource*; (ii) from the node that receives resource side, it is called *borrowing resource*. The resource's owner is the node which loans. The remaining resource is called *reserved resource*. Besides, Fig. 2 shows also how a *LM* constructs a comprehensive view of resource availability in the whole system. A *LM* uses this view to admit applications and control their execution on its node. A *LM* may exchange messages to request additional resources to other *LMs* when the available resources on its node do not meet application requirements.

Automatic processes increase the use of resources in the system: (i) at admission step if the available resources do not meet application resource requirements, the *LM* requests additional resources to other ones: this process is called dynamic borrowing of resources; (ii) when a *LM* does not use its resource borrowings during some periods, it releases automatically a percentage of these resources: this process is called resource release.

We call synchronization the invoked process when two parts of a distributed application simultaneously execute their steps using a shared resource, e.g., file transfer between nodes uses bandwidth. To avoid that several distributed applications attempt to use a shared resource at the same time, a pseudo protocol based on token is proposed. According to this protocol, shared resources are managed by the *LMs* collectively. Only one distributed application synchronizes its steps at a given time and this application must be triggered by a *LM* that has obtain the token. The node that holds the token manages also the queue of token requests according to a FIFO (First In First Out) principle. In this paper, a new version of this pseudo protocol based on EDF (Earliest Deadline First) is implemented. A token request contains a deadline time *t* that is the latest time to obtain the token in order to execute the related application before its deadline. This way aims to take into account emergency of applications and to avoid aborting applications when the request of an urgent application is at the end of the queue.

The previous work describes how to coordinate *LMs* in a decentralized system in order to better use resources and to execute applications before their deadline while reducing the number of exchanged messages. However, the approach had no support for automatically adapting application's behavior. This paper deals with adaptation issues. We build adaptation strategies upon our previous work and we show how to use our application model to coordinate adaptation in a decentralized system.

3. Requirements and Choices

3.1. Middleware Choice

In open environments, an execution support can assist the application to provide the expected QoS. However, diversity of applications, fluctuation of systems, heterogeneity of environments, computing costs and communication infrastructure make it complex to provide a support to each application. These difficulties force the designers to share an infrastructure called middleware for QoS management. Our middleware aims to face the complexity and heterogeneity inherent in distributed systems. It provides a common programming abstraction across a distributed system. This significantly reduces the burden on application programmers by relieving them of this kind of tedious and error-prone programming. The proposed middleware is a layer that copes with adaptation issues and controls application's behavior on each node. It cooperates with applications and schedules them based on run-time context. Any execution is decided by the middleware: the applications wait middleware orders before any step execution. Besides, applications run in a distributed context according to the following constraints: (i) any resource use should be declared, which means that applications all are registered; (ii) we assume that the underlying layers, including operating system and communication layers, supply all the required services to manage resources.

3.2. Adaptation Policy

There are two different approaches to deal with QoS. When QoS management is done apart from applications, for instance through process priorities, the approach is non-intrusive.

This approach is very attractive since all the applications benefit from the same services [3]. Nevertheless, the same service, e.g., delaying, will not affect applications in the same way, and in the worst case results are useless. In fact, only the applications have the necessary knowledge to set up alternative behaviors and to tune resources gracefully. In the intrusive approach adopted in this work, applications are specifically designed to participate with their environment (middleware, operating system, etc.), and to adapt their behavior.

3.3. Resource management

In open environments, QoS management systems deal with unpredictable events. Therefore, static scheduling does not apply, and we use dynamic scheduling solutions. These dynamic solutions aim to enforce the specified QoS properties while dealing with unexpected events that arise during execution. Resource reservation is required to ensure properties when event arrivals are unpredictable. Using the average load reservation policy, each activity declares an acceptable value of resource need within a period, for instance 25ms of CPU per second. When an activity asks for admission, the middleware accepts or rejects it according to resource availability. This policy assumes that activities organize themselves automatically: wait and synchronization are not managed, which is a major drawback for distributed applications. Planning reservation is a precise policy. Each task supplies its worst case resource requirements and its deadline. From these requirements, an exact planning is computed (e.g.,[7]). Before any task launching, the admission control checks resource availability. Planning reservation is harder to implement than average load reservation, but it takes into account synchronization, and ensures end-to-end deadline. In this paper we adopt a reservation planning.

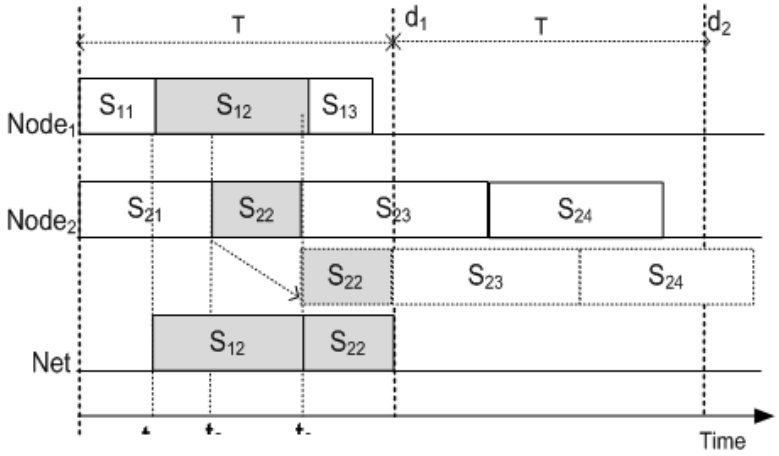

Figure 3. Synchronisation Conflict

Our approach is based on the use of approximate scheduling since it does not ensure the execution of the distributed applications. Fig. 3 depicts a conflicting situation in which an application should be aborted. This figure shows two distributed applications on nodes $Node_1$ and $Node_2$ with the schedule of their successive steps *S11*, *S12*, *S13* and *S21*, *S22*, *S23*, *S24* respectively and their deadlines d_1 and d_2 respectively. The gray steps (*S12* and *S22*) use a shared resource. The others steps use local resources only. These two applications need to synchronize their steps *S12* and *S22* with other nodes that are not represented Fig. 9 for the

sake of simplicity. The Net axis shows the used time of the shared resource for the two nodes. At time t_1, $Node_1$ requests the token and obtains it to execute step *S12*. While $Node_1$ uses the token, $Node_2$ in turn requests the token at time t_2 and waits for it. Finally, LM_1 releases the token at time t_3 and LM_2 obtains the token, to start the execution of *S22* on its node. In any case, LM_2 will not execute *S24* before its deadline. This situation is a consequence of our admission policy checks: to admit applications on a period basis does not ensure the order of execution in one period [9]. Generally, this situation occurs when the system is loaded, which increases the synchronized steps with close deadlines.

3.4. Application Model

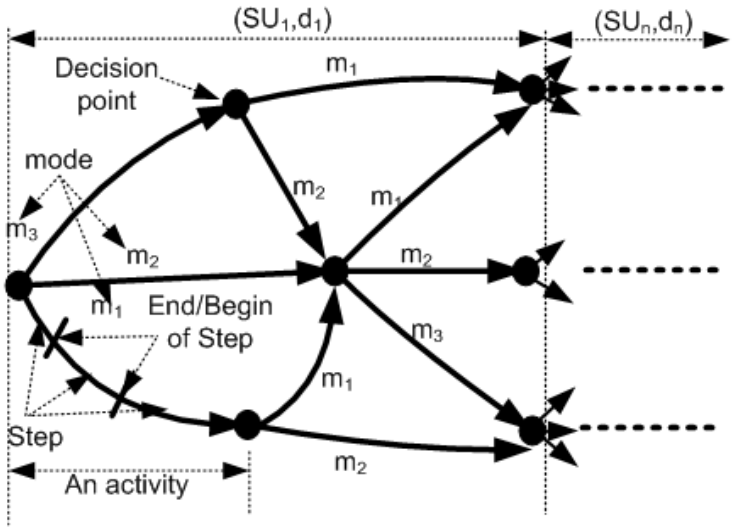

Figure 4. General Application Model

Our middleware holds data that are shared by all applications. In order to provide QoS support a general model for managed applications is needed. An application is designed as a graph (Fig. 4). In the graph edges represent the execution of activities.

An activity is a sequence of steps. Each activity (A_i) is associated with a mode (m_i) and a utility (U_i) that specifies the benefit related to the mode choice. A step is associated with resource requirements (e.g., CPU use, memory consumption). Vertexes are decision points, from which several behaviors provide the same service with different QoS levels. A local application is designed as one graph while a distributed application is designed as a set of sub graphs. The number of sub graphs is equal to the number of distributed application parts. Fig. 3 shows an example for a distributed application on two nodes $Node_1$ and $Node_2$. The whole application is a sequence of Scheduling Units. Each Scheduling Unit (SU_i) has a deadline di and is made up of a set of activities. A *SU* of the whole distributed application may concern one or more nodes, e.g., on Fig. 4, SU_1 on $Node_1$ and SU_1 on $Node_2$ belong to the same *SU* on several nodes while SU_j is a *SU* of the application on $Node_2$. When a *SU* is distributed on several nodes there is at least an activity that is distributed on these nodes. A distributed activity is designed as sub-activities on nodes with the same mode and utility, e.g., on Fig. 4, (A_1, m_1, U_1) on Node1 and (A_1, m_1, U_1) on $Node_2$ are parts of the same activity.

3.5. Application Example

Table 1. Application Example

Scheduling Unit	Activity	Mode	Step	Node
Start	A_Start	1	Load	Node1
	A_Start		Load	Node2
Sample	A_Acquisition	1	Acquisition	Node1
	A_Transmission	1	Transmission1	Node1
			Reception1	Node2
	A_Transmission	2	Compression1, Transmission2	Node1
			Reception2, Decompression1	Node2
	A_Transmission	3	Compression2, Transmission3	Node1
			Reception3, decompression2	Node2
	A_Display	1	Display1	Node2
	A_Display	2	Display2	Node2
	A_Display	3	Display3	Node2
Analyze		…	…	…

Table I gives an example of a distributed application between two nodes: part A on $Node_1$ and part B on $Node_2$. This application supplies three data transmission modes. After application loading (*Load*) on node $Node_1$ and $Node_2$ and data acquisition (*Acquisition*) on node $Node_1$, data is either directly sent (*Transmission1*) or compressed (*compression1* or *compression2*) before being sent (*Transmission2 or Transmission3*). Node $Node_2$ receives data directly (*Reception1*) or receives data and decompresses it (*Reception2* and *Decompression1 or Reception3 and Decompression2*). Finally, data is displayed either in mode 1 (*Display1*) or in mode 2 (*Display2*) or in mode 3 (*Display3*).

4. QoS Adaptation

4.1. Middleware Architecture

Our middleware is composed of a set of *LMs* on nodes. The underlying systems (OS and Network) are expected to provide all necessary services to *LMs*. Each *LM* manages applications running concurrently. A *LM* admits, schedules steps of activities and adapts the application's behavior based on information about applications as well as the execution context. A *LM* implements also capabilities for intra-node communication, e.g., between a *LM* and an application, and inter-nodes communication, e.g., between *LMs* for synchronization issues or adaptation coordination.

4.2. Execution Model

Numerous works deal with QoS management of distributed multimedia applications, e.g., [1][4][5][6] which are periodic, and based on continuous control of data flow, for instance, change in the video *frame* rate, generally during the next period [1], or feed-back adaptation in [4]. Our approach deals with applications stimulated by events whose arrival law is unpredictable, and change occurs at predefined decision points and at activity level. From these points, an activity is selected according to strategy; no change is possible between the steps.

4.3. Phases

In the sequel, we call admission phase when a *LM* decides the acceptation of a new *SU* of a current or new application in a system whereas during the execution phase the applications execute their steps under the control of the middleware. During these phases, the middleware performs adaptation in order to fit the current execution context. At admission phase, the *LM* selects a schedulable path in a *SU* according to the currently available resources. At execution phase, to perform adaptation, the *LM* tunes resources use by selecting the modes, which define the best feasible path in the application graph.

4.4. Utility

In our approach an activity is associated with a mode and a utility that assesses the degree of satisfaction of the user. The more the utility is the more the satisfaction is. Our middleware aims to maximize the overall instantaneous system utility *U*. This is similar to an optimization problem of resource use, which is NP-hard [12]. The *LMs* face to unpredictable events that make it impossible the search of an optimum. At the best, we could try to compute a succession of instantaneous optima but the sum of intermediate optimal decisions does not lead to a global optimum decision. Our middleware implements a heuristic algorithm based on EDF (Earliest Deadline first): when all the steps can be scheduled, this algorithm finds the solution while when the steps are not schedulable the solution is arbitrary. The *LMs* choose an activity according to the context to select the higher utility. Let's assume U(i) and $U_m(i)$ the selected and the maximum utility at decision point *i* respectively and *N* the number of decision points along an execution path. Obviously:

$\frac{U(i)}{U_m(i)} \leq 1$ .At each decision point the *LM* chooses an activity with a utility U(i) in such a way that $\frac{U(i)}{U_m(i)}$ tends to 1. Let $U = \frac{\sum_{i \in N} U(i)}{\sum_{i \in N} U_m(i)}$. U defines the execution quality of an application in the system. It is equal to 1 when all application activities are executed with the maximum utility, and lower than 1 otherwise. Our middleware aims to increase U and the number of admitted applications.

4.5 Adaptation Coordination

An application is a set of *SUs* (Fig. 4) that are admitted and executed sequentially before their deadline. All *SUs* of a local application are executed on one node and the application adapts its behavior according to the context on this node only. When a *SU* is distributed on several nodes, the application adapts its behavior according to resource availability on these nodes. The concerned *LMs* coordinate in order to execute distributed activities in the same mode. For instance, according to Table I, when on $Node_1$, the application compresses data in the mode 3, on $Node_2$, the application must decompresses in the same mode. Adaptation is guided by only one *LM* that meets a decision point of the current *SU*. When the mode is chosen, the *LM* broadcasts the execution mode of a distributed activity to concerned nodes.

4.6. Adaptation Strategies and Scheduling Outlines

The basic idea behind QoS Adaptation is to change application behavior when possible according to the execution context. Activities have different resource requirements. A utility

assesses the level of satisfaction of the provided QoS. In this context at least two strategies are conceivable: (i) at admission phase, the *LM* chooses a schedulable path in a *SU* with the lowest utility and increases this utility when possible at execution phase or/and; (ii) at admission phase, the *LM* chooses a *SU* with the higher utility possible and increases this utility when resources are available at execution phase. According to our model, the lowest utility corresponds to a degraded service. As a result, the latter strategy executes fewer applications with a higher utility. In our work, a basic principle:" is any degraded service is better than no service at all". Therefore, we have chosen the former strategy since it aims to increase immediate availability of resources and executes as many applications as possible.

Our adaptation policy is based on the following rules:

- Resource reservation occurs at *admission phase* to ensure resource availability. The SUs of applications are admitted in the schedulable modes with the lowest utility;
- Scheduling occurs at *execution phase*; a *LM* chooses a step to execute according to EDF based on WCETs;
- A step can be preempted by a *LM* in order to execute steps that use a shared resource when it receives a synchronization event;
- Time saved by actual durations less than WCETs as well as free resources are used to increase application utility.
- SUs and then Applications are rejected when available resources are not enough to run in the mode with the lowest utility. The utility (U) of any application that did not finish its execution is zero.

4.7. Application Description

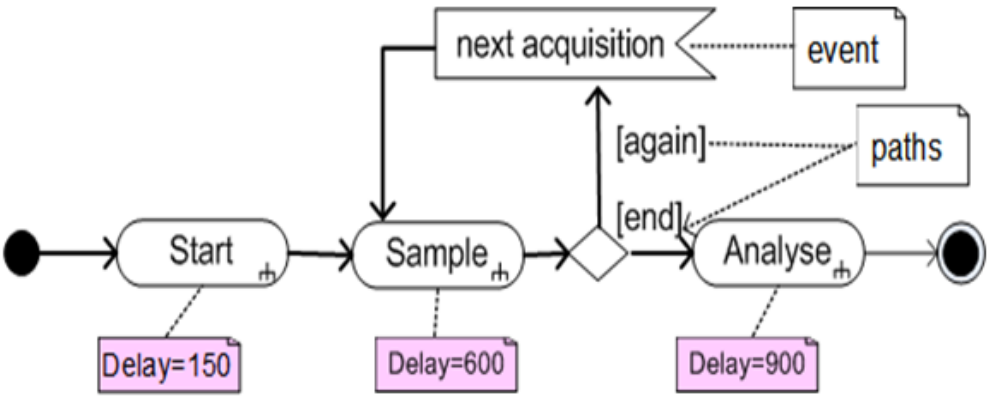

Figure 5. Global Execution Graph for the Application Table1

The easiest way to describe applications is to draw execution graphs adorned with QoS properties. The description lies on UML activity diagrams enriched with tagged values. At the higher level, an activity diagram describes the execution graph of the *SUs*: the Fig. 5 represents the higher level of the application given Table 1. The deadline comments define tagged values, i.e., attributes of SUs. Events and paths are simply described using UML notions.

Again, each *SU* is described using a hierarchy of activity diagrams. The Fig. 6 describes the SU Sample. In this diagram, UML activities are steps while activities are not represented explicitly. However, any path between a *mode-choice* and any *mode-choice*, initial/final node represents an Activity. The mode-choice is a stereotype that extends the metaclass

DecisionNode of UML, and it owns a specific icon. *Guards* on the outgoing edges are mode names.

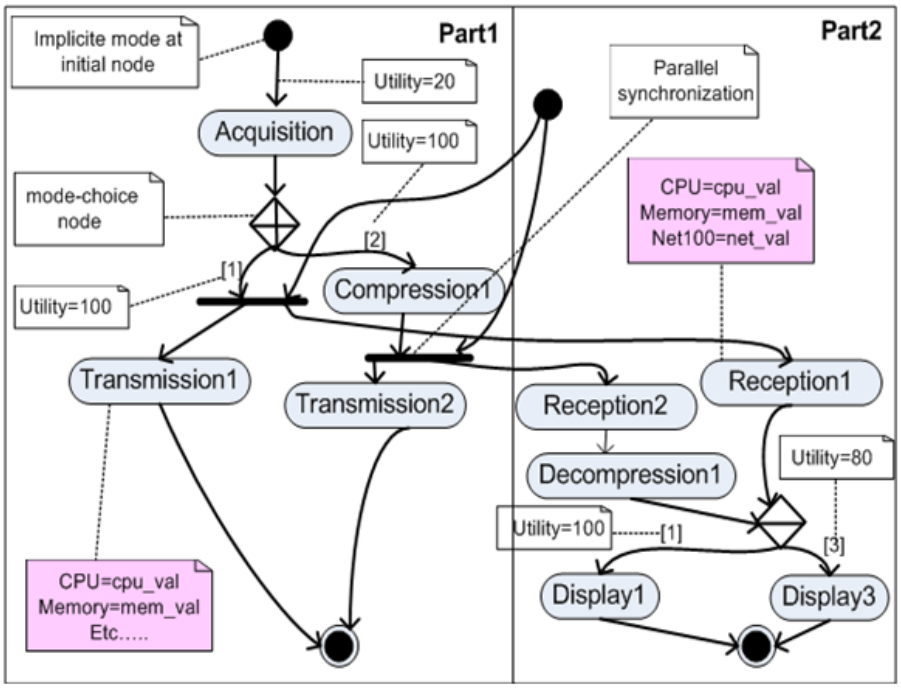

Figure 6. Execution Graph for the SU Sample

Parallel and sequential synchronization are described using join nodes and fork nodes. The swimlanes *Part1* and *Part2* represent the execution graphs for the two parts of the distributed application. For the sake of space, only two paths are drawn in this diagram and resources requirements are shown for *Acquisition* step only.

The two parts of the application run in parallel on their nodes. When LM_1 selects mode 2 with utility 100, after the execution of the step compression1, parallel synchronization ensures that the two nodes are ready to execute the activity in the next step.

5. Experimentation and Evaluation

This section describes the practical environment of our simulations. It explains the adopted technology and the platform in which our framework is implemented. As we experiment with a decentralized architecture, the use of autonomous entities such as agents seems suitable. We give below our definition of agent and we justify the choice of a platform that provides the necessary services to agents.

5.1. Agent Based Implementation

There are several definitions of software agent [13][14]. The term agent will be used to refer to a computer program that has the ability to take independent action, to react and to communicate. The agents communicate in the same host or over the network in order to coordinate their action. They can exchange simple or complex information.

A multi-agent system is a set of agents [15]. A multi-agent application consists of an environment and agents. The environment contains information that the agents use and manipulate to execute tasks.

5.2. JADE Agent Platform

To implement agent-based software, a multi-agent platform is required since it provides both a model for developing multi-agent systems and an environment for running distributed agent-based applications. Within the area of multi-agent systems, efficiency is a key issue. The efficiency of an agent system relies on both its design and the underlying agent platform. JADE platform [16][17] is widely known, easy to use, and well documented. The agent design of JADE and its Java implementation offer good runtime efficiency. Each JADE agent holds a collection of behaviors that are scheduled and executed to carry out agent actions. Every JADE agent runs in a Java thread, which satisfies autonomy properties, and all JADE behaviors are executed within a single Java thread. For inter-operability between agents or multi-agent systems, the JADE platform uses the FIFA specification [18]. Three mandatory roles are present in JADE: AMS (Agent Management System) that controls access and use of the platform, DF (Directory Facility) that provides yellow pages service, and ACC (Agent Communication Channel) that provides the message transport service. Containers may be used for separating groups of agents in a multi-agent system even when all agents run on the same host. The FIFA service agents (e.g., the AMS and the DF) are located in the main container that is always launched by JADE platform. JADE implements a distributed agent communication channel and provides synchronous and asynchronous agent communication.

5.3. Performance of JADE

The Choice of JADE is based on investigations made by numerous works on several existing multi-agent platforms performance [19][20][22]. According to these works, JADE is an efficient environment limited only by the standard limitation of Java programming language, which is interpreted and executed in a Virtual Machine: processor speed, amount of available memory and speed of network connection. The environment itself does not introduce substantial overhead. JADE can run experiments with thousands of agents distributed on several machines and communicating by exchanging several thousands of messages.

5.4. JADE Overhead

As any platform of agents JADE introduces overhead since it uses additional layers (communication protocols and others) to manage itself. Precautions must be taken to reduce the overhead during simulation [23], e.g., to group agents in the same container reduces the overhead of messages between agents. In our simulation, a container simulates a node and all agent applications of the node live in the same container. According to [19], the overhead increases when sending large size messages. Our *LMs* exchange messages but the content is a serialized class with few characters which reduces the overhead. However, due to JADE implementation, some actions can have a great impact on the total overhead, especially when change occurs at remote container such as agent creation, agent destruction, finding receivers agent location, finding agent service. All the above actions require messages between the container in which the action appears and the main container where live AMS and the DF. Thanks to the distributed caches in JADE agent platform, the overhead related to the agent location and agent service search is reduced. Besides, in our approach agents are long lived software objects. An agent thread is created for each application and deleted at the end of the execution. However, to assess CPU overhead on a node, we measured the average execution

times of all the middleware functions (agent creation, agent deletion, admission, message preparing time, message processing time, etc.).

In a decentralized system, messages may introduce latencies. However, in our approach, most messages are asynchronous. Synchronous messages appear when *LMs* synchronize their steps only.

As we simulate a multi-agent system, it is important to talk about the amount of data which are transferred on network. Although our messages size is few characters, the number of network messages plays an important role. Besides, adaptation strategies are built upon [9] in which an aim was to reduce the number of exchanged messages between the LMs in a decentralized system.

6. Results

To validate our approach, several simulations have been done. We consider several models of applications such as Table I. A *LM* (Fig. 7) consists of three agents. A Scheduler agent implements adaptation strategy and performs admission and scheduling tasks. The Scheduler makes decision based on execution context and information about applications (application models). It communicates (1) with other schedulers on remote nodes for synchronization issues or adaptation coordination. Before any execution of a step that uses a shared resource, it communicates (2) with the Global Resource Manager agent that implements mechanisms to manage shared resources; Shared resources are managed by the Global Resource Managers collectively (3).

On a node a Generator agent generates (5) applications according to arrival law as explained in the following subsection. On a node, agents use a common interface (4) to communicate with remote agents.

The applications are integrated into the agent architecture as follows: one single agent simulates local applications while several agents simulate distributed applications.

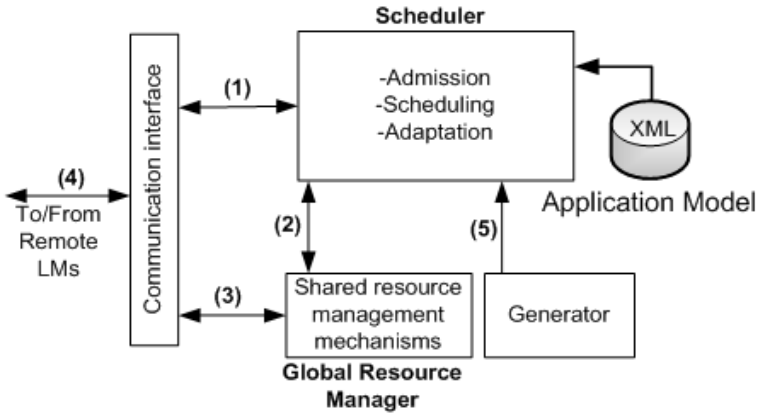

Figure 7. LM Functionalities

6.1. Experimental Conditions

Each *LM* holds a description of all the applications in the system. The results below were measured under the following conditions: (i) The deadline of each generated application is set to 2*SW + random [1*SW..2*SW], with SW = Sum of WCET (Worse Case Execution Time) of the application steps (path in the graph); (ii) The execution time of each step, is set to 2*WCET/3 + random [0..WCET/3]; (iii) The arrival law of applications is simulated by the

random time random [0..C], where C is a positive number: the smaller is C, the more applications arrive frequently and the more the system load increases; (iv) WCETs of applications' steps are defined in such a way that the sum of WCETs of all distributed applications is three time longer than the sum of WCETs of all local applications (applications on one node).

6.2. Measures

Fig. 8 shows CPU use. The wasted CPU (W/O axis) is due to aborted applications as shown the exemple Fig. 3. Overhead (O/W axis) is only local CPU and is due to agent creation/deletion time, admission time, creating/treating message time etc. The time needed to exchange messages is the sum of the time needed to process the message and the time needed to transfer the message over the network. In the estimation of the overhead, the time for creating messages and the time for processing confirmation messages are taken into account. The time needed to transfer the messages is not represented. On a 100Mbps network with three PCs (Windows XP professional, physical 512Mo) and JADE 3.3, the Round Trip Time average time for message with content from 500 to 4000B is 4ms.

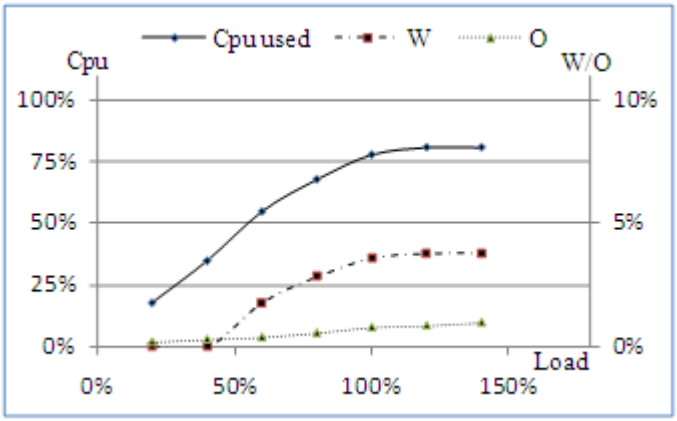

Figure 8 : CPU Use

Fig. 9 compares our adaptation policy (*Adapt*) and the best effort policy (*Best-effort*). Useful CPU time is equal to CPU used minus wasted and overhead time (Useful = CPU used – (wasted CPU + overhead)). It defines the effective time used by applications that completed their execution before their deadline. The Best effort policy admits all applications and set their behavior to the highest U. Therefore, the best-effort CPU use is greater or equal than in our approach. However, as best-effort does not check admission, many applications abort because they are not completed before their deadline, which explains why the useful time in Adapt is greater. Consequently Adapt is more efficient than best-effort.

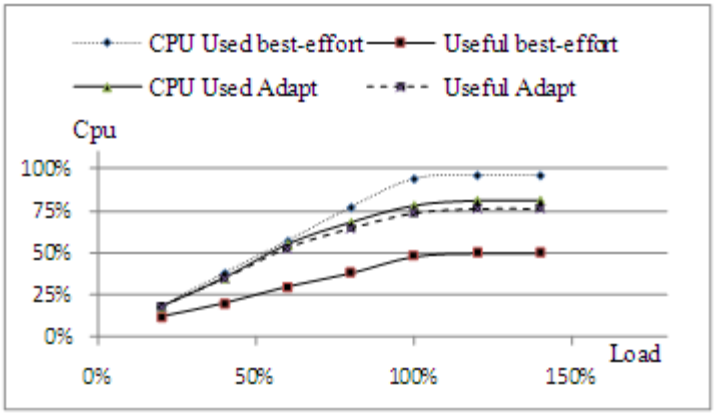

Figure 9. Best effort vs. Our Approach

Fig. 10 compares two approaches. The first is our approach without adaptation (*Max*). It admits the applications with the highest U when application is schedulable. The second is our approach with adaptation. It admits applications with the lowest U and increases the utility when resources become available (*Adapt*). The comparisons are based on the useful times and the utility U as defined in the section 4 (Utility). From about 45% the Adapt policy performs adaptation (U < 1) when the system increasingly becomes loaded while with the Max policy the applications are always executed with the highest utility (U = 1). On Fig. 10, there is little to choose between the two useful time curves. When the system is not loaded, the applications are executed with the same utility in the two approaches. When the system is loaded, there is always an application to admit. On Fig. 10, considering only the useful time and the utility, we could conclude that *Max* policy is efficient. But Fig. 11 gives the number of completed applications before their deadline.

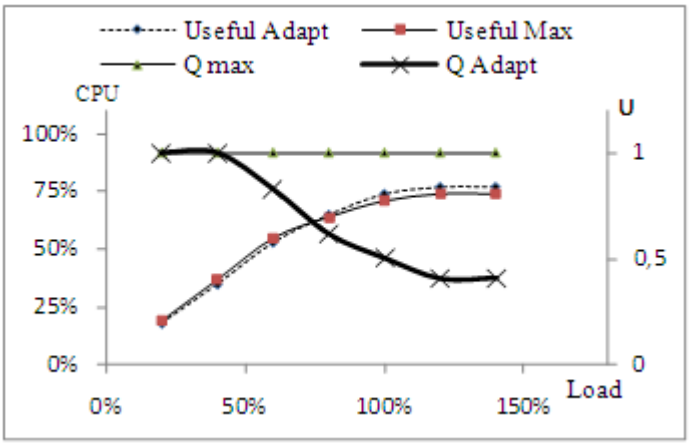

Figure 10 : Comparison of two Approaches

On Fig. 11, the Apps. axis indicates the number of completed applications before their deadline. Although the utility is lower in Adapt, the number of completed application in Adapt policy (*Completed_Adapt*) is greater than in Max policy (*Completed_max).* Besides, at

admission phase, *Max* rejects applications that could be executed when *Adapt* had been adopted. Hence, adaptation approach is more efficient.

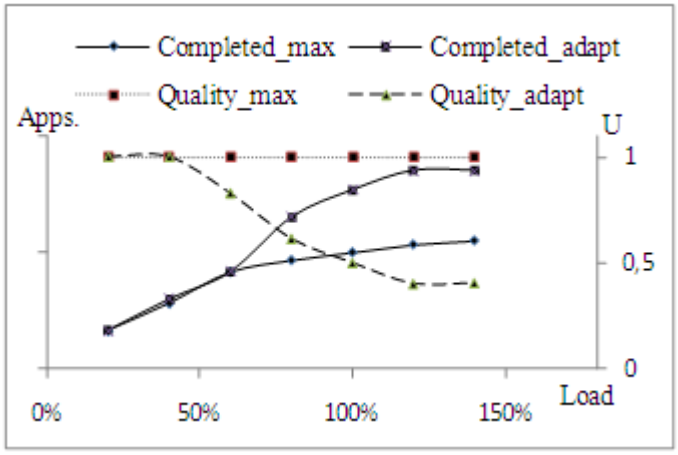

Figure 11. Number of Admitted Applications

7. Discussion

This section discusses some aspects since several alternatives decisions are conceivable at a given time.

- In our approach, the applications are inserted in a list before their admission. When an application reaches a decision point and if the available resource can admit another application, two situations are conceivable: we can choose to admit a new application or to adapt the current application. The first choice increases the number of admitted applications while the second increases the utility of the current application. As the already admitted applications provide an acceptable level of QoS since, according to our principle, to execute an application with the lowest utility is more profitable than nothing, we have chosen the first strategy. However, between the new *SU* of the already admitted application and the *SU* of the new application, we choose the first in order not to waste the execution time for the previous *SU* when after the admission of the new application the available resources do not meet the *SU* resource requirements.
- When an application reaches a decision point the *LM* may request additional resources (as explained in [9]) in order to improve the level of QoS of the current application. However, requests for additional resources require additional messages. When the system is loaded, these requests have little likelihood to be granted. Besides, in the decentralized system, latency is an important factor when an application has a deadline. For these reasons, there is no request for additional resources at decision points. Requests for additional resources are done at *admission phase*.
- In our approach a distributed application synchronizes its steps when a *LM* obtains the token. The distributed applications that need the use of a shared resource are queued in a list while the *LM* waits the token. When the *LM* obtains the token, it may choose to execute all waiting applications before releasing the token. This policy could avoid additional messages and reduces the total overhead since

messages are required to request again the token [9] after each execution when the *LM* releases the token. However, when an application on another node has a closer deadline and waits the token, its execution may be compromised. Besides, to abort an application is expensive [3]. In order to avoid aborting applications after the execution of the current distributed application, the token is sent to the node that has the more stressed application.

8. Related Works

Some works deal with periodic applications. [4] proposes a hierarchical, recursive, resource management architecture. It supports provision of integrated services for real-time distributed applications and offers management services for end-to-end QoS Management. This work changes the application behavior as a whole while in our approach adaptation is performed at activity level. [31] proposes a multi-agent-based, decentralized, adaptive QoS management framework. QoS management task is divided into subtasks. These subtasks are managed by the corresponding agents that collaborate or negotiate to perform adaptation at task level. Negotiation is done on a node to select a set of QoS for the stream in order to maximize the total utility under the resource constraint conditions. Inter-nodes negotiation aims to resolve QoS conflicts between the streams. In our approach, resource availability is constructed by each *LM* on its node in order to make local decision while avoiding negotiation which is expensive since our applications have deadline. These multimedia applications belong to periodic systems while our approach deals with aperiodic systems in which events arrival law is unpredictable. A periodic system is a particular case of aperiodic system.

Other approaches propose an intrusive policy and are based on application models [7] and [8]. The former proposes an efficient centralized approach aiming at adapting application behaviors. A unique scheduler tunes and schedules the use of the whole resources for all the concurrent applications in order to maximize the overall QoS of the supplied services. The latter integrates scheduling algorithms into reinforcement learning strategies in order to control resource use. It implements a centralized architecture. Our approach implements a decentralized approach and is based on an approximate scheduling.

[3] implements behavior adaptation based on web pages. Adaptation is performed by changing dynamically contents of Web pages. Each Web page has several versions with different QoS requirements. However this work is non-intrusive and concerns alternatives driven by the OS only, which limit its use. [24] describes how priority and reservation-based Operating System and network QoS management mechanisms can be coupled with adaptive middleware to better support distributed applications. However this solution is specific and concerns dynamic, distributed, real-time, and embedded applications with stringent end-to-end real-time requirements. In [25], two policies are distinguished: reservation-based system using reservation and admission control, and adaptation-based system attempting to adapt the system behavior to available resources. Our approach mixes these policies. The reservation-based policy ensures that an application is admitted when resources are available and the adaptation policy aims to increase the utility of applications according to context. In [26], authors propose a model for dynamic adaptation that can be customized by applications designers in order to satisfy adaptation needs. The model is based on a set of mandatory functionalities and an optional functionality. Adaptation mechanisms are associated to entities such as the whole application, processes, services, components or data and each entity is considered as composed of a set of components, leading to a hierarchical organization of adaptation mechanisms. In our approach, adaptation is associated to decision points only and

is performed at activity level, which is more flexible. [27] is based on the runtime environment and/or compilers to make applications adaptive. In our approach, the developers of adaptive applications define the behaviors of their applications at higher abstraction level. Adaptation strategies are not restricted by any environment and/or compiler. [28] proposes a reflexive framework to modify application behavior in the underlying runtime environment or in the adaptation framework itself. In our approach common adaptation mechanisms are in the middleware while specific mechanisms such as step execution at a decision point are included in the code. Others such as [29] and [30] describe analytic models to adapt applications using feedback. Our approach implements a heuristic and schedules steps based on their WCET (Worse Case Execution Time). Adaptation is performed at decision points to select an activity with the highest utility possible to fit the current context.

9. Conclusion

We have presented a framework for QoS management to cope with unpredictable variations of resources in distributed environment.

In order to provide QoS support, a general model of applications is described. This model exhibits several alternative behaviors of applications to provide the same service with different resource requirements and different quality. From this model, application designers may deal with application requirements according to their own policy.

To ensure QoS properties when event arrivals are unpredictable, resource reservation is required. We used resource reservation and a heuristic algorithm to schedule the resource use for all distributed applications.

Our framework manages the QoS of distributed applications by adapting dynamically their behavior to fit the current context. Adaptation relies on operating modes and is performed at predefined decision points where an activity with the higher utility is selected. Coordination between nodes ensures a consistent global adaptation.

Future works will focus on the refinement of the middleware functions. Besides, we plan to add a tool to help the designers to capture and evaluate QoS requirements and provide the required QoS specification. Finally, we also plan to attempt to take into account others application models and to include load balancing concepts.

References

[1] S. Brandt, G. Nutt, T. Berk, J. Mankovich, “A dynamic quality of service middleware agent for mediating application resource usage”, RTSS, pp.307-317, 1998.

[2] QoS-Basic Framework (ISO/IEC JTC1/SC21 N9309), 1995

[3] T.F. Abdelzaher, K.G. Shin, and N. Bhatti, “Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach”, IEEE Trans. Parallel and Distributed Systems, V 13, no. 1, p. 80-96, 2002.

[4] I. Cardei, R. Jha, M. Cardei, and A. Pavan, “Hierarchical Architecture for Real-Time Adaptive Resource Management”. Middleware'02, LNCS 1795, Springer Verlag, pp. 415-434, 2000.

[5] B. Ensink, V. Adve, “Coordinating Adaptations in Distributed Systems”. Proceedings of the 24th International Conference on Distributed Computing Systems, pp.446- 455, 2004.

[6] L. Abeni, G. Buttazzo, “Hierarchical QoS management for time sensitive applications”, Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium, On pp.63-72 ; 2001.

[7] P. Vienne, J.-L. Sourrouille and M. Maranzana, “Modeling Distributed Applications for QoS Management”,: 4th Int. Workshop On Soft. Eng. and Middleware, pp.170-184, 2005.

[8] P. Vienne and J.L Sourrouille, "A Middleware for Autonomic QoS Management based on Learning Software” Engineering and Middleware (SEM 2005)", ACM Proceedings, 2006

[9] N. Batouma, J.-L. Sourrouille" A Decentralized Resource Management Using a Borrowing Schema", Proceedings of the 8th ACS/IEEE International Conference on Computer Systems and Applications, AICCSA 2010.

[10] UML: Unified Modeling Language, http://www.uml.org

[11] F. Alhalabi, B. Narkoy, R. Aubry, M. Maranzana, L. Morel and J.-L Sourrouille " Centralized vs. Decentralized QoS Management Policy", 3rd IEEE Inter. Conf. on Information and Communication Technologies : From Theory to Applications, pp. 1-6, 2008.

[12] C. Lee, J. Lehoczky, R. (Raj) Rajkumar, D. Siewiorek, "On Quality of Service Optimization with Discrete QoS Options", *RTAS*, pp.276-286, 1999.

[13] S. Hyacinth, T. N. Divine, "A Perspective on Software Agent Research", The knowledge Engineering, Camb. Univ. Press, 99.

[14] S. N. Hyacinth, software agents: An overview, Intelligent Systems Research Advanced Application and Technology Department, knowledge Engineering Review, Vol. 11, pp 1-40, sept 1996

[15] N. A. Avouris, L. Gasser, "Distributed Artificial Intelligence: Theory and praxis" chapter Object Oriented Concurrent Programming and Distributed Artificial intelligence, Kluwer Academic Publisher, pp. 81-108, (1992).

[16] R. Leszczyna, "Evaluation of agent platforms". In: Performance, Computing, and Communications, IEEE International Conference on pp. 857- 864, April 2004.

[17] B. Kalle, G. Daniel and N. T. Simin, "Scale-up and performance studies of three agent platforms". IEEE Inter. Conf On Performance, Comp.and Communications, pp.857- 863, 2004.

[18] FIPA, http://www.fipa.org, accessed 22th may 2009.

[19] J. Kresimir, J. Gordon, K. Mario, "A performance Analysis of Multi-agent Systems", published in the journal : Inter. Transaction on Systems Sciences and Applications, Vol. 1, n. 4, 2006

[20] M. Luis, M. S. Jose, M. A. Juan, Performance Evaluation of Open-Source Multi-agent Platforms, AAMAS'06, ACM 2006.

[21] K. Chmiel, D. Tomiak, M. Gawinecki, P. Karczmarek, M. Szymczak, M. Paprzycki, "Testing the Efficiency of JADE agent platform", Proccedings of the 3th Inter. Symposium on Parallel and Distributed Computing, pp.49- 56, IEEE, 2004 .

[22] M. Berna-Koes, I. Nourbakhsh, K. Sycara ; Communication efficiency in multi-Agent Systems, Proceedings of the International Conference On Robotics and Automation, IEEE, Vol. 3, pp. 2129-2134, ,2004.

[23] E. Cortese, F. Quarta, G. Vitaglione, "Scalability and Performance of JADE Message Transport System", Centrino Direzionale isola F7, Telecom Italia, 2002.

[24] R. E. Schantz, J.P. Loyall, C. Rodrigues, D. C. Shimidt, Y. Krishnamurthy, and I. Pyarali. "Flexible and Adaptive QoS Control For Distributed Real-time and Embedded Middleware. Proc. of the ACM/IFIP/USENIX Inter. Conf. on Middleware 2003."

[25] B. Li, K. Nashrstedt, "QualProbes: Middleware QoS Profiling Services for Configuring Adaptaive Applications", LNCS 1795, pp.256-272, Middleware 2000.

[26] M. T. Segarra, F. André, "A Distributed Dynamic Adaptation Model for Component-Based Applications", Proceedings of the International Conference on Advanced Information Networking and Applications, IEEE, pp.525-529, 2009.

[27] S. Vadhiyar, J. Dongarra: Self Adaptability in grid Computing, Concurrency and Computation: Practice and Experience, Special issue: Grid performance, pp.235-257, 2005.

[28] J. Keeney and V. Cahil, "Chisel: a policy-driven, context-aware, dynamic adaptation framework." in 4th International Workshop on Policies for Distributed Systems and Networks, 2003.

[29] B. Li , K. Nahrstedt, "Impact of Control Theory on QoS Adaptation in Distributed Middleware Systems", In Proceedings of the 2001 American Control Conference, 2001.

[30] C. Koliver , K. Nahrstedt , J.-M. Farines , J. d. S. Fraga and S. A. Sandri, "Specification, Mapping and Control for QoS Adaptation", Real-Time Systems, Vol. 23, pp.143-174, Springer Netherlands 2004.

[31] M. Kosuga, T. Yamazaki, N. Ogino, J. Matsuda, "Adaptive QoS Management Using Layered Multi-agents System for Distributed Multimedia Application", Proccedings of the International Conference On Parallel Processing, pp.388-394. 1999.