Decentralized Resource Management Using a Borrowing Schema

Narkoy Batouma, Jean-Louis Sourrouille

Decentralized Resource Management Using a Borrowing Schema

Narkoy Batouma
INSA Lyon, LIESP
F-69621, Villeurbanne, France
narkoy.batouma@insa-lyon.fr

Jean-Louis Sourrouille
INSA Lyon, LIESP
F-69621, Villeurbanne, France
Jean-Louis.Sourrouille@insa-lyon.fr

***Abstract*— Resource management is a main issue when distributed applications should meet some Quality of Service (QoS) constraints. Centralized approaches are widely used for many good reasons, but decentralized approaches are often more reliable. A local view of the system from a node is not sufficient to make sound decisions. To synchronize their actions, nodes must exchange data with each other. Exchanging data incurs a cost; hence, to reduce the amount of exchanged data is desirable. This paper presents a distributed middleware to manage resources in a decentralized manner, using an approximate scheduling and a resource-borrowing scheme to improve the total QoS of the system. On each node, managers construct a comprehensive view of resource availability in the whole system. Our borrowing scheme aims to guarantee that as many applications as possible meet their deadline, to reject as few applications as possible, while decreasing the exchange costs and increasing the use of resources. Simulations show the usefulness of the approach and give encouraging results.**

Keywords-QoS; Resource Management; Borrowing Scheme; Scheduling; Decentralized Approach; Agent.

I. INTRODUCTION

Generally, Distributed Applications *DAs* are carried out in environments providing the best-effort policy. Unless constraints are specified, developers focus their effort on writing business code without taking into account QoS requirements, e.g., response time, resource allocation [1]. As long as resources are sufficient, the best effort policy is satisfactory, but when resources are scarce, control is needed. QoS-driven resource management is required to make the system more predictable [2], [3], to manage resources efficiently, and to provide QoS support for applications. Resources are managed in order to maximize an optimization criterion [4]. For example, we could want to execute a set of tasks as fast as possible or we could want to execute as many tasks as possible within a given period. The general problem of scheduling is known to be NP-Hard [5]. Anyway, when the arrival law of applications is unknown, only some transient optimal solution can be reach, which has no real interest. To cope with complexity, heuristics provide approximate solutions [6]. The scheduler represents an abstraction level that decides on the tasks' ordering in order to maximize some optimization criterion. This scheduler is part of a resource manager that dynamically reschedules task to maximize the QoS that is provided by the whole system. The architecture of a distributed system has a significant impact on its provided QoS. Among the numerous architectures aiming to implement resource management [7], [8], we distinguish between centralized and decentralized scheduling. In the centralized approaches, scheduling is made by one scheduler according to the entire system state [6], [9] ; in the decentralized approaches [10], [11], [12] the decision is constructed by several schedulers which hold a partial knowledge of the system only and deal with consistency and communication explicitly [10], [13], [14]. We intend to experiment with decentralized architectures. Our approach does not relate to large systems such as grid computing [15] but relies on a Local Area Network with a "flat" architecture. Our approach does not either relate to resource management such as virtualization [16], which introduces an abstraction layer on top of resources to hide physical characteristics from the users. Besides, applications are carried out on the nodes to which they are admitted. Although resource use on each node depends on the context and resource availability, our approach does not implement load balancing [17] in the sense that applications do not move.

In this paper, we address distributed applications whose arrival law is unpredictable and systems subjected to QoS constraints such as the value of any service provided after the deadline is null. In this context, continuous run-time adaptation of application behavior, e.g., graceful degradation is a promising approach. To adapt resource consumption, applications participate in resource management. As a drawback, applications are specific, but this intrusive policy [1] is the only way to tune resources finely and consistently. We need a middleware that fully controls resource use based on a decentralized approach. The middleware is the layer that copes with resource management issues and controls applications' behavior on each node. We have divided the works into two parts. The first part proposes the decentralized approach of resources management. The second aims to describe adaptation of distributed applications. This paper deals with the first part only. Scheduling uses a scheme of resource borrowing to make it possible to keep locally a global view of system resource availability, allowing local managers to make decisions while reducing communication costs.

The paper is organized as follows: first, we present the problem description (section II) and the application model that will be used to simulate our approach (section III). Then, we describe our approach (section IV) and show how the system is implemented in practice (section V). Finally, we give some results (section VI) and we compare our approach to related works (section VII).

II. PROBLEM DESCRIPTION

The choice between centralized and decentralized architecture depends on needs. Each one has its advantages and drawbacks [7], and represents solutions from different perspectives. We experimented with several centralized policies [6], [9]. Centralized architectures are efficient but have drawbacks suck as inflexibility, scalability, single point of failure, bottleneck. This paper does not intend to compare these architectures but to investigate a decentralized architecture. In any decentralized approach, communications are crucial for schedulers to coordinate properly: communication is the way to obtain additional information from each other scheduler. Generally, it is unrealistic for the schedulers to keep up to date the resource global states continuously. A decentralized approach has to deal with communication explicitly, in particular in dynamic environments when communication incurs a cost or when continuous communications is not feasible. We wish to experiment a system in which the key characteristics are: first, schedulers are decentralized (a scheduler on each node), and second, they share information in order to improve the overall QoS of the system. In this context, each scheduler has to maximize both local and global resource use and has to minimize communication costs. Our main aims are: (i) To reduce the number of exchanged messages between nodes as much as possible; (ii) To improve the performance of the system by decreasing wasted time as much as possible; (iii) Once application is admitted and scheduled, to guarantee the execution as much as possible.

III. CONTEXT AND AIMS

A. Environment

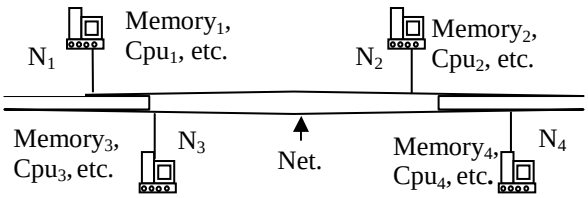

Figure 1. Environment Description

This section describes our environment and our application model. A *Node* is a separate piece of hardware that has at least one processor as in UML [18]. The environment (Fig. 1) is made up of a set of nodes, N_1, N_2, N_3, etc. that are connected to each other via a network. Each node has its own resources (Cpu_i, $memory_i$, etc). There are two types of resources: local resources are associated with a node, and managed by a *Local Manager (LM)* on this node; shared resources such as bandwidth are managed collectively by all *Local Managers.*

To manage efficiently resources, the middleware must hold a complete description of both available resources in the environment, and resources required for application execution. Since applications and middleware cooperate in the resource management process during execution, a protocol defines how applications communicate with the middleware. For an application to be accepted in the system, an admission step is necessary to check resource availability and to decide whether the application may be executed before its deadline.

B. Resource Management

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 refuses 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 more precise policy. Each activity supplies its worst case resource requirements and its deadline. From these requirements, an exact planning is computed (e.g., [6], [19]). Before any activity 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, which is a major property: any admitted activity is guaranteed to execute (to cancel a running task is always expensive [20]). In this paper, we choose the Planning reservation policy.

C. Application Model

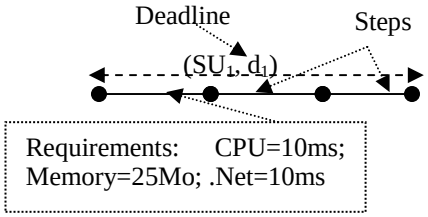

Figure 2. Application Model

To control their behavior, applications are designed especially to be managed by the middleware. An application is viewed as a succession of steps (Fig. 2), and its execution is a path. At the end of step execution, i.e., at each vertex, applications inform their manager and wait an order to resume, implementing a Master/slave paradigm. Each step is associated with resource requirements.

A Scheduling Unit (SU) is a sequence of steps with a deadline. An application is built of one or more SUs. To execute an application is to carry out these different SUs while respecting their deadline. When steps of different parts of a DA communicate, they should be synchronized. For example on Fig. 3, the parts App_{11} on N_1 and App_{12} on N_2 of an application include a data transfer. When App_{11} and App_{12} arrive respectively at vertex 1 and 2, called rendezvous points, a synchronization between the steps Transfer and Reception occurs. After the end of the synchronized step, i.e., at vertexes *3* and *4*, each part executes separately until the next synchronization or the end of application.

D. Overview of Centralized policies

Our team has carried out several experiments about centralized architectures. Within the centralized policy, one scheduler called *Global Manager* holds the full description of

all applications and of the environment in which applications execute. This unique scheduler tunes and schedules the use of the whole resources (local and global resources) for all the concurrent applications in order to maximize the overall quality of the supplied services. On each node, there is a *Local Manager* that acts as an intermediary between applications and the unique global scheduler, which is hosted by a particular node. In [6], the global scheduler uses a heuristic to build a precise planning and orders local scheduler to execute application. In [9], a learning approach is used to control resource use.

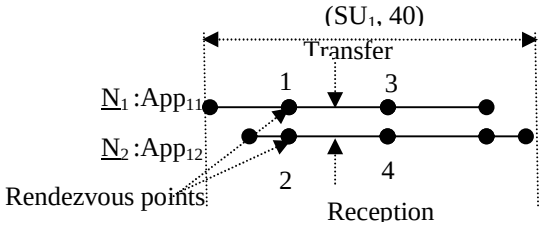

Figure 3. Synchronization between steps

IV. THE PROPOSED BORROWING APPROACH

This section gives an overview of the proposed approach to enable resource management. In our decentralized architecture, each node owns a scheduler (called *LM*) which deals with a partial observation of the system and makes local decision. In the sequel, the use of any resource is represented by its time duration on a time axis. On each node, a *LM* deals with local and shared resources management. Let N_1, N_2, N_3 be nodes with their resources respectively Cpu_1, Cpu_2, Cpu_3 and a network symbolized by *Net*.

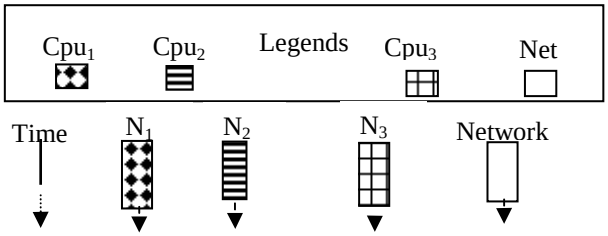

Figure 4. Resource use

Fig. 4 represents the use of resources for each node along an axis of time. It is important to notice that an application has various resource use durations according to node properties such as processor speed. Using a simplistic approach to the problem, each node is associated with local resources, and the *LM* has a local view of its resources only. Distant resource availability is unknown, i.e., the node N_1 does not know resource availability on N_2. When a Local Application (*LA*), i.e., which uses only local resources, arrives, a local view of resources is enough to make a decision about admission. However, when a DA, which uses resources on different nodes, arrives, the *LM*, which manages resources on each node, admits the application without guarantee about execution, i.e., using a local view of resources only: this is a kind of best-effort policy. In our approach, local and global resource states are available on each node, which aims to guarantee end-to-end execution of applications.

To illustrate our purpose, suppose the following points:

- The time is subdivided into periods of length **T**, and on each node resource use is expressed as a percentage of **T**;
- At the beginning of a period, each *LM* reserves a part of its resources and loans the remainder to others *LMs*;
- Shared resources are managed in the same way.

Fig. 5 depicts resource availability percentage for two successive **T** assuming an equal percentage of the shared resource *Net* for each node. Each *LM* holds a quota of available resources on other nodes in addition to its own resources. To admit a new application, *LMs* request additional resources when available resources do not meet application requirements.

The novelty of this scheme is to keep locally a view of resource availability within the whole system. Using our approach, *LMs* construct this view to minimize messages between nodes. The following sections detail this approach. First, it is necessary to define the sense of *Loaned resources* and *borrowed resources* as shown on Fig. 6. In a period **T**, nodes N_1 and N_2 hold resources Cpu_1 and Cpu_2 respectively. On each node, the *LM* keeps one part of its resources and loans the remainder to other *LM*s. The resources that each *LM* distributes to the others are called *loaned resources*. On the other hand, the resources that each *LM* receives are called *borrowed resources*.

A. Static Borrowing and Type of Resources

According to our approach, at each beginning of period resources are associated to each node. We distinguish two types of resources on each node:

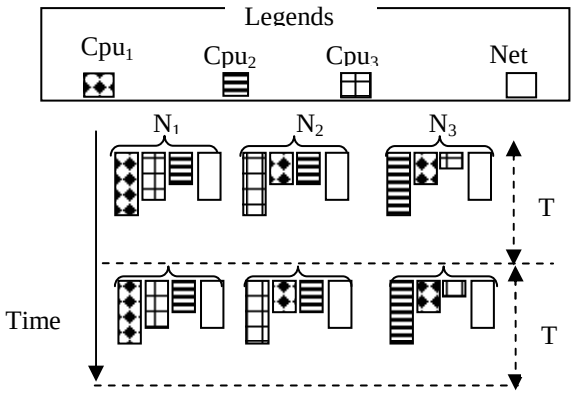

Figure 5. Resource percentage in the proposed approach

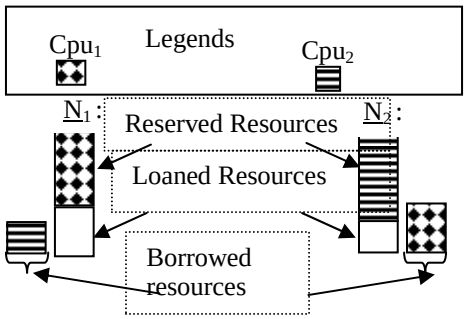

Figure 6. Loaned and borrowed resources

- *Reserved resources* are local node resources that each *LM* reserves for the execution of its applications. When a *LA* arrives, its required resources must meet the reserved resources of the node to accept admission. On the other hand, when a *DA* arrives, the required resources of the application local part, i.e., that will be carried out on the local node, must meet the reserved resources.
- *Borrowed resources* are the resources loaned by one *LM* to the others. They give a local representation of remote resources availability. When a *DA* arrives, the required resources of the distant parts, i.e., that will be carried out on the remotes nodes, must meet the borrowed resources for each node as well as common resources to be admitted.

Notice that reserved resources are used by both *LA*s and *DA*s that arrive on a node, while borrowed resources are used by *DA*s only. To exchange between nodes the network resource *Net* is used jointly with other resources since when N_1 needs to transfer data to another node, it uses the resources Cpu_1 and the network *Net*. Of course, the receiver node also uses CPU resources. These mechanisms induce the following properties for each period **T**:

- For each resource, the sum of reserved and loaned resources is equal to **T**;
- The sum over all nodes of shared resources is equal to **T**;
- The sum of times of scheduled step resources is lower than or equal to T since there not necessarily always an application to execute;

Each *LM* can set its loaned and borrowed amount of resources. Each node can specify its own borrowing policy that is made available for all *LM*s.

The following subsection describes different phases performed by the middleware to manage resources.

B. Admission and execution phases

The proposed middleware uses two phases to manage resources: admission and execution. The admission phase decides whether an application should be admitted into the system after checking available resources. This phase aims to guarantee end-to-end execution. When its admission succeeds, the application is planned and waits order to execute. During the execution phase, on each node the *LM* selects the next step to execute based on scheduling decisions. At any time and on any node, only one step is running. Thanks to this mechanism, *LM*s control resources used by applications. To control application execution, the middleware uses different messages:

- From *LMs* to applications: to start the execution of a step, to load an application, to kill application, etc.
- From the applications to the *LMs*: to notify the end of the current step execution and messages related to the arrival such as request for admission.
- From one *LM* to another *LM*: to notify application arrival and admission or to synchronize applications.

It is important to notice that in this scheme, the order to execute steps is not known at admission phase since *LMs* check resource availability only. This order is set at execution phase, when a *LM* must choose the next step to execute according to the scheduling algorithm. This way, newly admitted applications with their own deadline that force the order are taken into account.

During admission phase, when the available resources are lower than application resource requirements, a *LM* can request additional resources to achieve admission. The following subsections describe the process.

C. Dynamic borrowing or loan

Dynamic borrowings or loans are invoked by *LMs* to increase their current quantity of available resources at admission phase. These requests of additional resources belong to one of the following cases:

- When a *LA* arrives on one node, if there is a lack of reserved resources, the involved *LM* can request resources it has loaned to other *LM*s. It can ask to one node or even several when the quantity is significant. This procedure decreases the amount of borrowed resources from some remote nodes;
- When a *DA* arrives on a node, if borrowed resources are insufficient, the involved *LM* can request resources to increase its borrowed resources and admit the new application. This procedure decreases the available local resources of the requested remote nodes. In this case, the *LM* can also request resources it has loaned to other *LM*s.

Suppose that a *DA* between nodes N_1 and N_2 arrives with resource requirements respectively Cpu_1 and Cpu_2. N_1 tries to admit the application. If the available amount of Cpu_2 does not match application requirements, N_1 can ask additional Cpu_2 to N_2. This request can be granted or not by the other *LM* according to resource availability. Throughout execution, *LM*s can formulate several requests for the same resource.

D. Borrowings policy evolution

In the precedent sub-section, we explained how an *LM* obtains additional resource. These processes modify the borrowing policy (i.e., they increase resources of some nodes and decrease those of others). We define here the resource owner as the manager that has lent the resource. Besides, borrowing policy can also change along the time. Two processes are proposed: (i) After a time $\Delta\mathbf{t}$ ($\Delta\mathbf{t} \leq \mathbf{T}$) from the beginning of the period, if one *LM* does not use a borrowed resource, its owner may recovers it after agreement; (ii) After $\mathbf{k}$ ($\mathbf{k} \geq \mathbf{1}$) periods, if one *LM* does not use a borrowed resource, its owner automatically recovers $\boldsymbol{\alpha}$ percent from this resource where $\mathbf{k}$ and $\boldsymbol{\alpha}$ are implementation parameters.

E. Application Deadlines

The middleware manages applications made up of *SU*s having deadlines. In theory, all deadlines would be accepted but as resources are managed on a per period basis, what to do when deadlines are within the period, e.g. $\mathbf{d_4}$, $\mathbf{d_1}$ and $\mathbf{d_3}$ (Fig.7)?

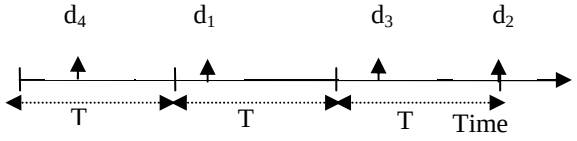

Figure 7. Applications deadlines

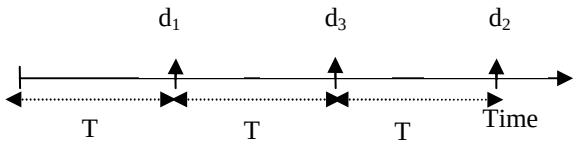

Figure 8. Deadlines alignment

Since within a period the execution order is unknown, it is not possible to guarantee deadlines within the period. As a result, any deadline is aligned on periods (Fig. 8): any deadline of application d_i is rounded up to $\mathbf{aT} < d_i$, with **a**, an integer and **T** the period. Any deadline lower than the period such as $\mathbf{d_4}$ on Fig. 7 is rejected.

V. IMPLEMENTATION

A. Choice of Agent technology and platform

To implement a decentralized approach, an agent-based approach provides a suitable design and implementation framework for our purpose. Moreover, the software agents' technology is largely used in complex systems such as resource management in strongly dynamic distributed environments. Within the area of multi-agent systems (MAS), efficiency is a key issue. The efficiency of an agent system relies on both its design and the underlying agent platform. Many agents' platforms exist based on works such as [21] [22]. We choose JADE, 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 cooperatively within a single Java thread. For inter-operability between agents or a multi-agent system, the JADE platform uses the FIFA specification [23]. Jade provides asynchronous agent communication. The agents are grouped into containers that are independent of each other. Containers can be distributed on several hosts or on a same host, which makes it possible to simulate several nodes or only one node on the same computer. Moreover, JADE provides powerful graphic tools to manage its environment, for example, a Snifter agent to trace the exchanged messages between nodes, and a tool to remotely manage agents (deletion, creation etc.). Finally, the ACL (Agent Communication Language) makes the communications between agents easier.

B. Shared Resource Management

To avoid use conflicts of shared resources such as bandwidth, a token mechanism controls shared resources. The node that holds the token can allocate shared resources to its applications. This pseudo protocol is based on the following points:

- Initially within the system, one node only holds the token, and the other nodes know which node holds the token.
- The node that holds the token uses a shared resource without any notice to other nodes. When another node requires shared resources, it should request the token. It sends a message to the node that has the token and waits while executing any other step that does not require shared resources. When the node receives the token, the pending step can use shared resources.
- When the token holder receives a request for the token, several cases are possible: **i**) If the node does not use the token, it releases it to the request sender; **ii**) If the node is executing a step that requires the token, the request is queued. When the token is free, the node processes the requests according to the FIFO principle. The queue is sent to the node that requested first the token; **iii**) If a request arrives after the token transfer, the request is forwarded to the new token holder.
- A node that receives the token notifies all the other nodes except the previous holder.

According to this scenario, to obtain the token in a system with **N** nodes, **1** message is sent to request the token, **1** message is sent for the reply, and (**N-2**) messages are sent to notify the others nodes, which allow updating the address of the new token holder. In total, **N** messages are necessary to obtain and use the token. Thus, *LMs* using this pseudo protocol manage shared resources collectively.

C. Scheduling Outlines

To get rid of scheduling complexity, the proposed algorithm is a heuristics based on EDF (Earliest Deadline First) due to its good properties: it is very simple to implement, and when there is a solution to the scheduling problem, EDF leads to this solution. When there is no solution, the result value of EDF scheduling is unknown. Our admission checks aim to reduce failures and in most cases, there is a solution. The planning is modified each time a new event occurs, e.g., admission, arrival of token and end of step. As soon as a node is idle, the next scheduled step starts. Our heuristics guarantees the execution of any admitted application except when there is synchronization as depicted on Fig. 9. This figure shows two *DA*s on nodes N_1 and N_2 with the schedule of their successive steps S_{11}, S_{12}, S_{13}, S_{14}, S_{15} and S_{21}, S_{22}, S_{23} respectively, and their deadline d_1 and d_2 respectively. The gray steps S_{12} and S_{22} use a shared resource. The others steps use local resources only. These two applications need to synchronize their steps S_{12} and S_{22} with others nodes that are not represented Fig. 9 for sake of simplicity. The *Net* axis shows the use time of the shared resource for the two nodes. At time t_1, N_1 requests the

token and obtains it for the step S_{12}. While N_1 uses the token, N_2 in turn requests the token at time t_2 and waits for it. N_1 releases the token at time t_3 and N_2 obtains the token, assuming that it is the first in the queue. In any case, N_2 will not execute the steps S_{23} before the deadline d_2. This situation is a consequence of our admission checks: to admit applications on a per period basis does not ensure the order of execution in one period. Generally, this situation occurs when the system is loaded, which increase the synchronized steps with close deadlines.

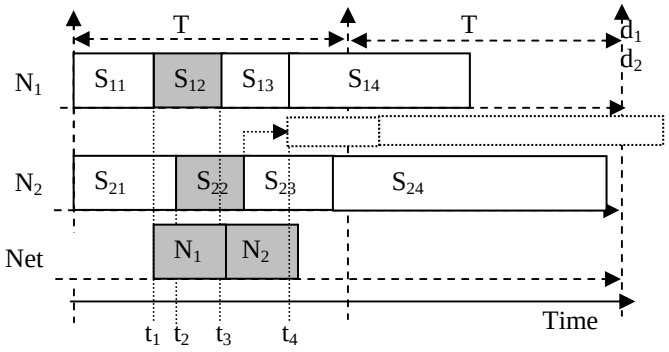

Figure 9. Conflict synchronization

D. Implementation choices

Some implementation choices are considered:

- Each *LM* manages a queue of pending applications. Admission occurs at the beginning of each period. When there is no raining step, admission may occur inside the period;
- Regarding *DA*s, the node that has registered the first part of the *DA* manages the admission;
- As *DA*s have parts on several nodes, the node managing the admission notifies about admission result the nodes having the other applications parts. This procedure allows planning application loading as soon as possible.
- Each *LM* executes steps according to its scheduling. When the step is synchronized, a message to plan synchronization is sent to the *LM* that has the corresponding synchronized step.
- Any application whose admission does not succeed is rejected.
- Steps using shared resources have a higher priority than step using local resource only. Shared resources are regarded as very scarce.
- When a step has started, its execution continues until the end except when a synchronization message arrives. In this case, the current step is preempted until the end of synchronization.

VI. RESULTS

To illustrate our purpose, several simulations have been done. On each node, there are a Generator Agent (*GA*) and a *LM* agent. The *GA* creates applications according to an arrival law (see subsection B). The *LM* implements our borrowing scheme: it deals with application admission and controls application execution. Each generated application sends a message to the *LM* to notify its arrival. The *LM* registers the application and its description, getting information (deadline, resource requirements, etc.) back from the description file. Admission of *DA*s start only when all theirs parts are arrived and are registered on all nodes, whereas admission of *LA*s start as soon as the application arrives and is registered. Once admitted, the application waits order to execute while the LM schedules the steps of the admitted *SU*. When a *DA* is running, each application part runs on its node. For example in a data transfer application, a part reads and sends data, while the second part receives and displays it. Applications are integrated into the agent architecture: one agent class implements behaviors of *LA*, while several agents' class implement *DAs'* ones. Thus, the middleware creates and manages agents, which represent applications. JADE allows creating a container for each node. Each container manages several agents.

A. Experimental conditions

For each *LM*, a description of the environment and all the applications is available. The results below were measured under the following conditions: (i) The deadline of each generated application is set to 2**SW* + *random* [1..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 more *C* is small, the more applications arrive frequently and the more the system load increases; (iv) *WCET*s of applications' steps are defined in such a way that the sum of *WCET*s of all *DA*s is three time longer than the sum of *WCET*s of all *LA*s.

B. Measures

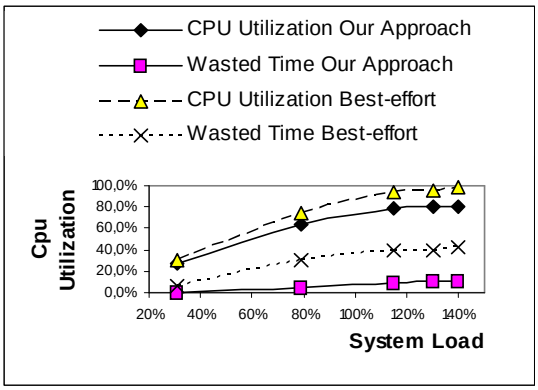

Figure 10. CPU utilization

Fig. 10 compares our approach and the best-effort policy. Results show the CPU utilization and the CPU wasted due to applications that are killed when they do not meet their deadline. In the proposed approach, the percentage of used CPU is not equal to 100% because the admission phase rejects applications whose requirements do not match resource

availability. The wasted time is due to applications that were admitted but finally were killed as in Fig. 9. The best-effort policy admits all the applications, which explains that the used CPU is greater but also that more applications are killed resulting in a larger wasted time than in our approach. CPU use reaches some maximum in both approaches.

Fig. 11 shows the effective use of CPU for both approaches. The effective use is measured as CPU use less wasted time.

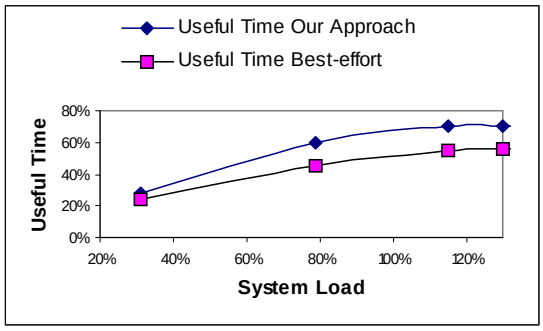

Figure 11. Effective use of CPU

The effective use of CPU is lower using the best-effort policy, because many applications are killed before their end, as soon as they violate their deadline, and any time spent to execute an application that will be killed is wasted. It is possible to reduce wasted time in our approach by applying strong admission checks which reduce the number of admitted and killed applications, but reduce also CPU effective use.

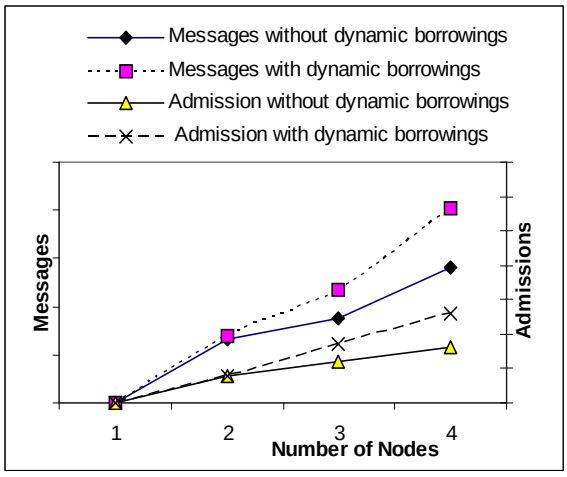

Figure 12. Messages and admissions function of nodes

Fig. 12 shows the number of admitted *DA*s and the number of exchanged messages according to the number of nodes with and without dynamic borrowing, i.e., process to ask additional resource at admission time. Dynamic borrowing increases the number of exchanged messages but also the number of admitted *DA*s. This result was expected because dynamic borrowings concern only *DA*s, hence favors the admissions of *DA*s. Since dynamic borrowing increases message cost, we could imagine mechanisms of automatic release of resources, i.e., at the beginning of each period, if a node does not use some resources, it might release them but no release is implemented in this experiment.

Fig. 13 illustrates the influence of the percentage of borrowed resources in our approach for a load of 60% and for three nodes. The use of dynamic borrowing increases the number of admitted *DA*s, hence the total number of admitted applications (*LA*s and *DA*s). The percentage of borrowing resources of x% means that each *LM* loans x% of its resources to each other *LM* and keeps the remaining. When this percentage increases, borrowing resources increase while reserved resources decreases. Fig. 13 also shows that when the percentage of borrowings reaches 40%, there are less admitted applications. That is explained by the fact that, the amount of remaining resources (reserved resources) is too small to admit both *LA*s and local parts of *DA*s, and requests for additional resources (dynamic borrowing process) are sent when the reserved resources are sufficient only.

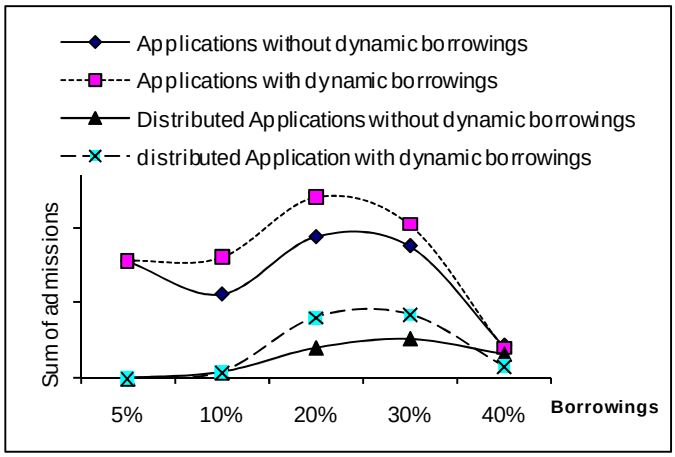

Figure 13. Admitted applications according to borrowings percentages

These results show that, when procedures of dynamic borrowing are implemented, the percentages of resource borrowing must be judiciously chosen. When this percentage is too much (40% for example on Fig. 13) or too less (5% for example on Fig. 14), less applications are admitted. But, when the percentage is reasonable (between 15 and 25% on Fig. 13), the probability to admit applications increases. Furthermore, if *DA*s are scarce into the system when *LA*s are frequent, the percentage of borrowings resource must be very low to avoid an important number of rejected *LA*s during admission because the *LA*s use reserved resources only and these resources increase when the borrowings decrease. In practice, it is not convenient that the *LM* loans a great part of its resources to the others and reserves a small part.

VII. RELATED WORKS

Recent works tackle the problem of coordination of multi-agent system in an uncertain environment while aiming to avoid wasting resources by minimizing the number of communications [10], [24], [25]. The approach in [10], for example, maintains agent coordination at run-time and chooses

to communicate only when there is a perceived benefit to team performance. This approach based on POMDPs (Partially Observable Markov Decision Problems decision) manipulates concepts such as state, action, transition, observation or believe in an uncertain environment and is difficult to implement. Our approach implements an easy to use borrowing system to manage resources. For distributed systems, [26] proposes a fully decentralized approach, with a job-shop scheduling based on a distributed combinatorial scheduling technique, to accommodate a large number of tasks with reduced messaging requirements. The developed approach relates to the deterministic systems whereas in our approach, we face the applications whose arrival law is unknown. [13] Addresses the problem of communication costs for a decentralized control in a multi-agents system. A formal model is presented using an extension of POMDPs and cooperative agents are able to share information at the off-line planning stage as if they were centrally controlled and were acting in real-time in a decentralized manner. In our approach, although the managed applications are fully described, the decisions are taken in real-time and in a decentralized way since no information about application is known before its arrival. The approach in [27] uses also a borrowing technique to support real-time interactive multimedia in the domain of wireless networks and mobile. [11] proposes a decentralized solution to monitoring hosts in a distributed system using multicasting facilities. Although multicast is an interesting alternative for dynamic borrowing requests, for example to ask a group of nodes some additional resources, as a drawback one multicast message generally induces several answers, which are not always beneficial for the whole system.

VIII. CONCLUSION

We have developed a decentralized approach aiming to manage QoS of distributed applications. Our main contribution is a borrowing scheme for resource management to execute applications while controlling resources. Using borrowing mechanisms, *Local Managers* construct and keep locally a view of local and remote resource availability to control admission of distributed applications. Our simulation shows that the effective use of the CPU is lower using the best-effort policy than our approach. Besides, dynamic borrowing increases the number of admitted applications. Our future works will deal with adaptation of distributed applications and will improve this approach simulating a variable period **T**. Further, some extensions such as grids and/or hierarchical system will be investigated.

REFERENCES

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

[2] K. Lakshmann and R.Yavatkar; "Integrated CPU and NetworkI/O QoS Management in an Endsystem"; IFIP 1997.

[3] A. Hafid; "Providing a scalable video-on-demand system using future reservation of resources and multicast communications"; Communications", Proc. 5 th IWQOS'97.

[4] S. French, Sequencing and Scheduling. Willey(1982)

[5] M. Pinedo: Scheduling: Theory, Algorithms and Systems.Prentice Hall (1995)

[6] P. Vienne, J.-L. Sourrouille and M. Maranzana, Modeling Distributed Applications for QoS Management. In Software Engineering and Middleware: 4th Int. Workshop. Springer, 2005, pages 170-184.

[7] F. Alhalabi, B. Narkoy, R. Aubry, M. Maranzana, L. Morel and J.-L. Sourrouille" Centralized vs. Decentralized QoS Management policy" Information and Communication Technologies: From Theory to Applications. ICTTA08. 3rd Int. Conf. on Vol., April 2008 Page(s):1 – 6

[8] P. Xuan and V. Lesser, Multi-agent policies:From centralized ones to decentralized ones. In *International Joint Conference on Autonomous Agents and Multi-agent Systems*, 2002.

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

[10] R. Maayan, S. Reid and V. Manuela, "Decentralized communication strategies for coordinated multi-agent policies"; Multi-Robot Systems. From Swarms to Intelligent Automata. Volume III, 93-105. 2005; Springer.

[11] M. Stumm," Strategies for Decentralized Resource Management"; ACM SIGCOMM Computer Communication Review; 1987, P. 245 – 253.

[12] M. Arora, S. Das, and R. Biswas, "A Decentralized Scheduling and Load Balancing Algorithm for heterogeneous Grid Environment", Proc. Of the ICPPW, Vancover, IEEE, Canada, 2002.

[13] V. G. Claudia and Z. Shlomo, "Optimizing information exchange in cooperative multi-agent systems" ; ACM, International Conference on Autonomous Agents. Proc. of the 2th int. joint conference on Autonomous agents and multi-agent systems. P.:137 – 144; 2003.

[14] R. Hino, K. Izuhara and T. Moriwaki," Message exchange method for decentralized scheduling", Assembly and Task Planning, Proc. of the IEEE Inter. Symposium on**,** 2001, On page(s): 244-249.

[15] K. Krauter, R. Buyya, M. Maheswaran, "A Taxonomy and Survey of Grid ResourceManagement Systems for Distributed Computing"; International Journal of Software Practice and Experience (SPE), Vol. 32, Issue 2, 135-164, Wiley Press, 2002.

[16] H. Sung and S. L. Jong, "Service Agent-Based Resource Management Using Virtualization for Computational Grid", Computational Science ICCS 07, LNCS 4488, pp. 966-969, Springer Berlin Heindelberg 2007.

[17] D. Grosu, A. T. Chronopoulis, "Algorithmic Mechanisms Design for Load Balancing in Distributed Systems"; IEEE transactions; 2004

[18] UML, http://www.uml.org, accessed 22th may 2009.

[19] E.M. Atkins , T.F. Abdelzaher , K.G. Shin, "Planning and Resource Allocation for Hard Realtime, Fault-Tolerant Plan Execution", Proc. Int. Conf. on Autonomous Agents, ACM Press, 1999, pp.244-251

[20] T.F. Abdelzaher, K.G. Shin, N. Bhatti, "Performance Guarantees for Web Server End-Systems:A Control-Theoretical Approach", IEEE Trans. Parallel and Distributed Systems, Vol. 13(1),2002, pp.80-96

[21] R. Leszczyna, "Evaluation of agent platforms". In: Performance, Computing, and Communications, 2004 IEEE International Conference on, pages: 857- 864. April 15-17, 2004.

[22] B. Kalle, G. Daniel and N. T. Simin, "Scale-up and performance studies of three agent platforms". Performance, Computing, and Communications, 2004 IEEE Inter. Conf. 2004 ; On page(s): 857- 863.

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

[24] D. S. Bernstein, S. Zilberstein and N. Immerman,"The complexity of decentralized control of Markov Decision Processes". In *Uncertainty in Artificial Intelligence*,2000.

[25] D. V. Pynadath and M. Tambe, "The communicative Multiagent Team Decision Problem: Analyzing teamwork theories and models". 2002. *Journal of AI Research*.

[26] S. Logie, D. Sabaz and W. A. Gruver ; Sliding Window Distributed Combinatorial Scheduling using JADE; Systems, Man and Cybernetics, 2004 IEEE International Conference on **;**, Vol.: 2, On p. 1984- 1989.

[27] E. K. Mona, O. Stephan and A. W. Hussein, " A Rate-Based Borrowing Scheme for QoS Provisioning in Multimedia Wireless Networks"; IEEE Transaction on Parallel and Distributed System, Vol. 13; No. 2; February 2002.