Efficient Implementation of Hierarchical Resource Control for Multi-agent Systems

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

For the Degree of Master of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

by

Xinghui Zhao

Permission To Use

In presenting this thesis in partial fulfillment of the requirements for a Post-graduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5C9

Abstract

Development of the World Wide Web makes it possible for multiple computers to work together in order to solve problems and make the most efficient use of resources. A distributed system is composed of such computers which are separately located and connected with each other through a network. One paradigm for computation using distributed systems is the multi-agent systems, in which many autonomous agents interact with each other to solve problems. The agents in a multi-agent system may be distributed on different computers (or nodes), where each computer owns its resources[1]*. Although the resources in a multi-agent system are connected by a network through which mobile agents can migrate for accessing sufficient resources, how to share these independently owned resources in both an effective and an efficient way is not fully understood. A key challenge in multi-agent systems is how to account for and control the resources which are located on individual nodes.*

The CyberOrgs *model offers one approach to manage resources among competitive or collaborative agents by organizing computations and resources in a hierarchy. A cyberorg encapsulates agents and resources in a boundary and distributes the resources available to it within this boundary. A cyberorg contained in another cyberorg has a contract with the outer cyberorg, according to which it receives resource that it may use. A cyberorg also encapsulates an amount of the* eCash*, which is the currency for purchasing resources from its host cyberorg. Therefore, cyberorgs have a hierarchical structure, where the resources flow down from the root to the leaves of this hierarchy and the eCash flows up from the leaves to the root of this hierarchy. However, the hierarchical structure of the CyberOrgs model presents challenges in scalability. As a result, efficiency is an important concern in the implementation of CyberOrgs.*

In this thesis, an efficient implementation of the CyberOrgs model is described. System design, APIs of the implementation, example applications, experimental results, and future directions are presented.

[1]Here by "resources," we mean computational processor time, memory, resources such as network bandwidth.

Acknowledgements

I would like to express my sincere appreciation to my supervisor, Professor Nadeem Jamali, for his invaluable guidance, suggestions, and encouragement. Professor Jamali offered tremendous help in my research work and during the whole writing process of my thesis. This thesis would be impossible without his knowledge, support, and patience. Professor Jamali is not only a good researcher, but also a considerate person that I always feel comfortable to talk with whenever I encounter any difficulties in my personal life. I really had a great time working with him. What I have learned from him will definitely keep benefiting my whole academic career.

Many thanks go to Professor Raymond Spiteri and Professor Kevin Schneider, the two professors in my thesis committee. Professor Spiteri managed to have meetings with me when he was very busy, and he shared his insightful ideas and advices with me. He made me realize that the application part of my research work could be very interesting, and worth putting more effort on. Professor Schneider provided very helpful comments and feedback which improved the quality of my thesis.

I would like to thank Professor Michael Bradley, for his willingness to be the external examiner in my thesis committee.

Department of Computer Science at University of Saskatchewan offers a friendly environment and sufficient utilities for the graduate students. I am very thankful that I could work in this wonderful environment. Special thanks go to our office staff, especially Ms. Jan Thompson, who offers mother-like love to all graduate students in our department, which makes our life much easier.

I would like to thank my colleague and friend, Chen Liu, for her help. She read my thesis and gave me useful suggestions, and she also helped me improve my sometimes precarious grammar.

I express my gratitude to my parents. My father always encourages me to make effort for achieving my goals, and he is not only a very thoughtful and supportive father, but also a good friend of mine. My mother taught me how to be an honest

and faithful person. I always feel her love and blessing, although she is no longer with us.

Finally, my deepest appreciation goes to my husband, Dr. Jian Wu, who is willing to offer his unconditional support to me at anytime and anywhere. He encourages me to conduct my research, even if we have to live separately in two different countries for a long time. He manages to accommodate my studies without any complaint ever. I would not be able to concentrate on my research if it was not for his unwavering love and belief in me.

Table Of Contents

List of Tables

List of Figures

List of Acronyms

MAS – Multi-Agent System

RPC – Remote Procedure Call

CCS – Calculus of Communicating Systems

AA – Actor Architecture

API – Application Program Interface

RM API – Java Resource Management Application Program Interface

JVM – Java Virtual Machine

RMS – Rate Monotonic Scheduling

EDF – Earliest Deadline First Scheduling

LOF – List of Figures

LOT – List of Tables

Chapter 1

Introduction

The field of multi-agent systems is a relatively new research area in computer science with most advances happening in the last 20 years. In this period, coordination among multiple agents has emerged as a key challenge in exploiting the possibilities presented by massive open distributed systems [6].

In a multi-agent system, the computations which are carried out by the agents are fueled by resources, but inadequate resource control may adversely affect the progress of computations in the system. Therefore, resource management is an important part of the coordination challenge.

1.1 Multi-agent Systems

An *open system* [16] is a system which is open to interaction with the environment. Computing entities may arrive at or leave an open system at any point of time; they may have different computing objectives, and the only requirement for those computing entities is the ability to communicate with each other. The concept of open systems has become increasingly relevant since when it was developed, because it is becoming ever more necessary for separately and independently developed systems to communicate in order to cooperate with each other.

Multi-agent systems [41] offer a natural programming abstraction for realizing open systems. A multi-agent system is composed of multiple interacting computing entities, know as agents, which are active objects that can migrate across machines.

There is no universally accepted definition of the term *agent*. In this thesis, we use the definition presented in [42]: *An agent is a computer system that is situated in some environment, and that is capable of autonomous action in order to meet its design objectives*. There are two key characteristics of agents. First, agents are autonomous [17], which means agents can decide for themselves what they need to do in order to achieve their goals. Second, agents are capable of communication [9], through which they may cooperate with each other so that it is possible to solve problems that are beyond the capabilities of an individual agent.

1.2 Resources in Multi-agent Systems

Multi-agent systems require different types of computational resources while pursuing their computation objectives, such as processor time, memory, and network bandwidth. Although resources are located on individual machines, when those machines are connected by a network, it is possible for agents to access the distributed resources. In order to get enough resources for a computation, agents may migrate from machine to machine, and obtain resources they require to achieve their goals.

In this thesis, we focus on *resource bounded* multi-agent systems, where resources are limited. A resource bounded multi-agent system can be viewed as executing in a space, and there are limited resources existing in this space. The computations carried out by the agents are executed in the space and share the resources in it. The fact that available resources are bounded results in competition among the agents, which may lead to uncertainty in the system [36], potentially threatening optimal execution of computations. Hence, it is critical to account for resource usage and coordinate resource access by agents.

1.3 Approach

Many approaches have been used for managing resources among competing or collaborating agents. In this thesis, we focus on hierarchical resource control approach

based on the *CyberOrgs model* [19], and its implementation which provides decentralized resource control in multi-agent systems. Conceptually, in CyberOrgs, distributed resources and agents are organized in a hierarchy. The components of this hierarchy are cyberorgs, which serve as encapsulations of computations and available resources. The root cyberorg of this hierarchy possesses all available resources, and each of the other cyberorgs obtains resources from its host cyberorg; each cyberorg divides its available resources between its own agents and its client cyberorgs according to a local *resource allocation policy* and the contracts with the clients. Resources eventually arrive at their destination cyberorgs, where they are consumed by the agents in order to support their computations.

Hierarchical control presents challenges in scalability. Imagine a large scale multi-agent system, where there might be thousands of agents working concurrently. The resource control hierarchy of this system may be extremely deep, and if so, the overhead caused by the hierarchical control would be prohibitive. To illustrate that, we use processor time control as an example. In a large scale multi-agent system, if we implement the processor time resource control using a hierarchy, we have to traverse the entire path from the root cyberorg to the leaf cyberorg in order to schedule a computation agent which is in that leaf cyberorg, and this traversal results in a very high overhead. Therefore, although hierarchical structure is an expressive format for resource control, it is difficult to implement it efficiently.

1.4 Contribution of this thesis

The approach that is presented in this thesis provides a scalable and efficient mechanism for implementing hierarchical resource control. We enforce hierarchical control using a physically flat implementation of the schedule with exactly the same control specification as the hierarchical one. In this way, we avoid the high overhead in implementing hierarchical control.

Our implementation of CyberOrgs extends *Actor Architecture* [25], which is a Java-based framework implementing the *Actor* [1] model of concurrency. We only fo-

cus on the processor time control in this thesis, but this approach can be generalized to control some other types of resources.

1.5 Outline

The rest of this thesis is organized as follows.

Existing work in related areas is reviewed in Chapter 2. System design and specific implementation issues are described in Chapter 3. In Chapter 4, we describe the application program interfaces (APIs) developed for the CyberOrgs model as well as some sample application programs which illustrate how to use the APIs. Experimental results are presented in Chapter 5. We present future directions in Chapter 6. Finally, conclusion is presented in Chapter 7.

Chapter 2

Related Work

2.1 Introduction

In this Chapter, the existing research work in related areas is reviewed. Section 2.2 presents the background work in object-oriented concurrency, especially Actor theory. Because resource management is the area that this research focuses on, approaches in resource management are reviewed in Section 2.3. Previous work in CPU resource allocation is reviewed in Section 2.4.

2.2 Models of Concurrency

Models of concurrency are used to formalizing concurrent computations in open systems. In this section, background knowledge about models of concurrency is reviewed.

$\pi-$calculus [33] is a calculus for expressing processes with changing structures. $\pi-$calculus was extended from the process algebra CCS (Calculus of Communicating Systems) [32]. The Actor model [1] is another model of concurrency for modeling concurrent and asynchronous processes.

We focus on the Actor model, because it offers a natural programming framework for implementing object-oriented distributed systems. In this section, concepts about the Actor model are presented, and some specific implementations of the model are reviewed.

2.2.1 Actors

Carl E. Hewitt first used the term "actor" in his early work for PLANNER [14], and he proposed the concept of actors in his paper [15] in 1977. Irene Grief developed an abstract model [13] for actors, and afterwards William D. Clinger developed the semantics for actors in his thesis [5]. Gul A. Agha extended actors to both a programming language [1, 3] and a data abstraction [2] for concurrent open systems.

Actors are autonomous computational entities which communicate with each other using buffered, asynchronous, and point-to-point messages. An actor encapsulates a state, a number of methods (which can change the state of the actor), and a thread of control. Actors are distributed over time and space. Every actor has a globally unique mail address, and it maintains a queue of unprocessed messages it has received. Figure 2.1 shows the structure of an actor.

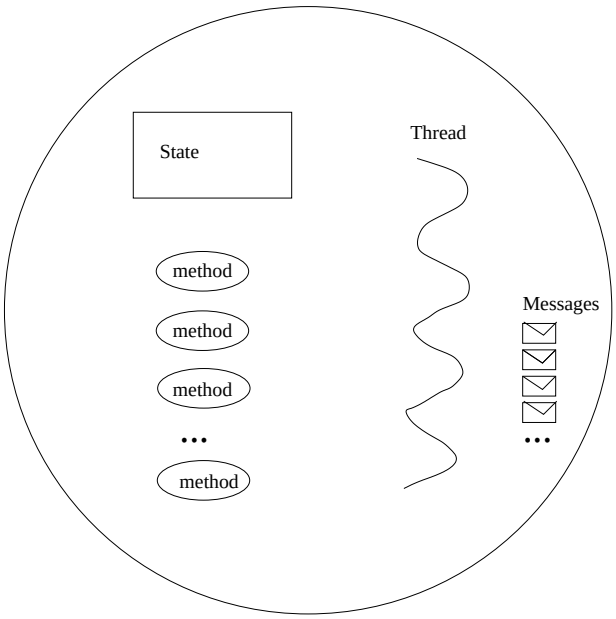

Figure 2.1: Actor Structure

The messages in an actor's message queue are processed one by one according to the order of arriving. While processing a message, three types of *actor primitives* may occur:

- Create finite number of new actors with some predefined behaviors. The cre-

ator actor knows the addresses of the new actors.

- Send messages to other actors. An actor can send message to another actor only when it knows the mail address of the destination actor.

- Change the actor's own state and be ready to process the next message.

The messages that are sent by actors eventually arrive at the destination actors, but there is no guarantee about the specific order of arriving.

2.2.2 Actor Systems

There are several implementations of Actors. Here we present two of them: *Actor Foundry* [24] and *Actor Architecture* [25].

2.2.2.1 Actor Foundry

Actor Foundry is a Java-based framework for the Actor model. An instance of the Actor Foundry run-time system is a *foundry node*, and there can be many actor instances in one foundry node. Asynchronous messages can be sent from one actor to another, and the message delivery in Actor Foundry is *weakly fair*[1]. The behavior of actors in Actor Foundry can be defined by programmers.

The structure of a foundry node is shown in Figure 2.2. Each foundry node consists of seven interacting functional units:

- *Actor Manager*:

  Actor manager is the central unit for every foundry node, and it is responsible for all inter-node communications. It is also responsible for carrying out the actions necessary to create an actor and begin scheduling it.

- *Actor Implementation*:

  An actor implementation represents an instance of the actor class. It transfers method calls to the actor manager which is in charge of the local actor actions.

[1]Here by "weakly fair," we mean that once a message is sent, it will eventually be delivered. Moreover, once a message arrives, it will eventually be processed

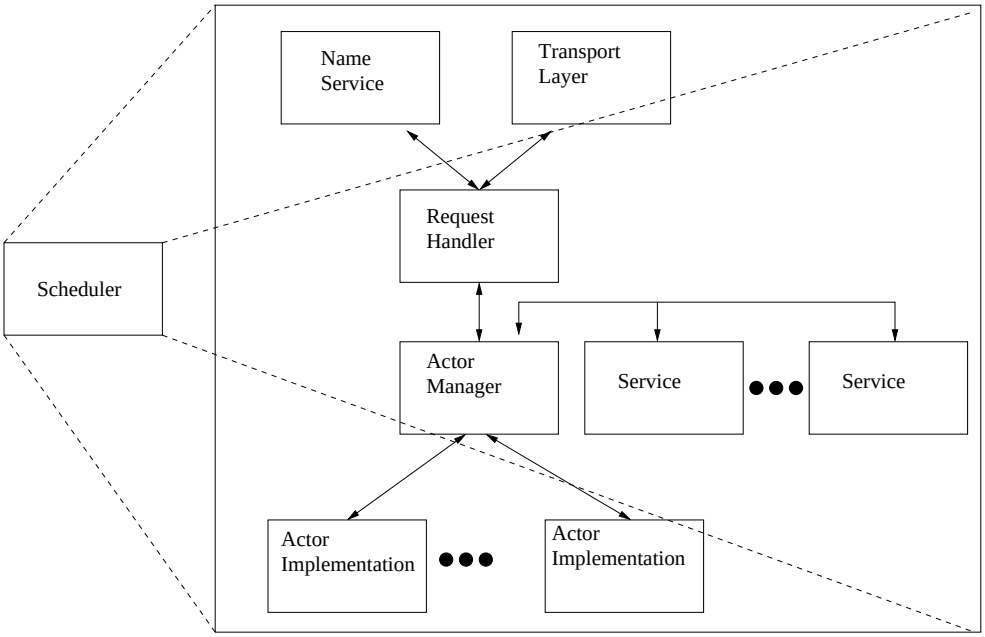

Figure 2.2: Actor Foundry Node Structure ([4])

- *Service*:

  The Service module provides additional services to local actors through the Actor Manager class. In order to access these additional services, an actor must communicate with the Actor Manager class.

- *Request Handler*:

  The Request Handler module plays the role of a connection medium between the local actor manager and actor managers on other foundry nodes. It provides remote procedure calls (RPCs) which are either synchronous or asynchronous, and it also provides access to the Name Service.

- *Name Service*:

  Every actor has a globally unique name, and all the names are generated by the Name Service module. A name is bound to the corresponding actor, and the Request Handler delivers the remote procedure calls according to these bindings.

- *Transport Layer*:

  The Transport Layer provides low level communication protocols. Messages

of any size can be transported and delivered.

- *Scheduler*:

  The Scheduler module is responsible for scheduling all the threads in a foundry node, and all threads are scheduled fairly.

2.2.2.2 Actor Architecture

Actor Architecture is another Java-based middleware system which provides an actor execution environment. An instance of Actor Architecture run time system is called an Actor Architecture platform (AA platform). Each AA platform consists of four service layers:

- *Message Transport Service*:

  The Message Transport Service is the communication interface of an AA platform. All messages that are sent to or received from another AA platform pass through the Message Transport Service layer.

- *Message Delivery Service*:

  The Message Delivery Service layer is responsible for handling all the local messages in the AA platform.

- *Actor Management Service*:

  The Actor Management Service manages the states of all actors and all the migrations on the AA platform.

- *Advanced Service*:

  The Advanced Service layer provides middle actor services, such as matchmaking and brokering.

The distributed resource management system that we present in this thesis has been developed by extending Actor Architecture. We chose Actor Architecture because all basic actor primitives are already implemented, and the structure of Actor Architecture is explicit, making it is easy to extend.

2.3 Resource Management

A multi-agent system can be constructed using the concurrent computation models we reviewed in Section 2.2. Resource management in a multi-agent system is critical, and it presents challenges in scalability and efficiency. In this section, several types of resource management approaches are reviewed.

2.3.1 Language Approaches

Ether [23] was one of the earliest languages to address resource allocation among concurrent components. In Ether, every process needs a *sponsor* that is assigned to it to support computations. Later on, ACORE [31], a concurrent programming language based on the actor model, incorporated the idea of Ether. There are *sponsor actors* in ACORE. The sponsor actors can process requests, and *ticks* are required in the process. A similar idea was used in Telescript [10], in which the computational resources are abstracted as *teleclicks*, and processes need teleclicks to accomplish computations.

Java [12] is a language which supports distributed applications by addressing portability, but Java does not provide adequate support for resource management. Many approaches towards resource management try to address Java's deficiency, such as JRes [8], JSeal2 [40], and Java Resource Management API [7].

2.3.1.1 JRes

JRes [8] provides an interface for accounting and limiting access to different types of resources, such as CPU time, network bandwidth, and main memory. JRes was implemented using both Java bytecode editing and native code to account for resources without changing the Java Virtual Machine [28].

CPU time is controlled in JRes through a mixture of bytecode editing and native code. When a new thread is created in the system, a native code routine is invoked so that an operating system level handle can be generated for this new thread in order to query the information about CPU usage in the system. A special thread as

part of the CPU accounting module uses the handle to query the operating system about resource consumption. As a result, the accounting for CPU time usage in JRes relies on the underlying operating system. Accounting heap memory usage in JRes is based on bytecode rewriting. The code of every method is modified by inserting appropriate bytecode before the instructions of allocating every object in the original code. Specially, for an array object, a native method is still necessary to be inserted after the allocating of array object in order to detect the deallocation of the array. For controlling network bandwidth resource, JRes relies on native code. Because in Java the methods that can access network resources are located in the java.net package, and these methods are protected or private so that they cannot be called outside java.net package. The author of JRes changed the java.net package to their own in order to keep track of the amount of data transferred by every thread.

The unit for resource control in JRes is the individual thread. However, using threads as the resource management units makes the accounting more difficult, because threads may access multiple program components and the resources obtained by one thread may be consumed by another thread. Therefore, the limitation of JRes is that it cannot handle object sharing between threads.

2.3.1.2 JSeal2

JSeal2 [40] also focuses on resource accounting like JRes, and the API of JSeal2 is similar with JRes too. The developers of JSeal2 were influenced by research on *resource bounded actors* [20], which was the early work of CyberOrgs.

In JSeal2, the basic unit of resource management is a *Seal*, instead of an individual thread. A seal may be either a mobile object or a service component, and each seal executes in a protected domain and shares no state with other seals. Seals in JSeal2 are organized in a hierarchy as the cyberorgs in CyberOrgs model. When the system starts up, the first domain, *RootSeal*, possesses all of resources that were obtained by Java Virtual Machine from the underlying operating system. Whenever a new child seal is created, the creator seal assigns some part of its own resource to the new seal, which generates a hierarchy of resource allocation: basically all of the

resources are owned by the rootseal, and a parent seal is responsible for distributing resources among its sub-seals.

The most important feature of JSeal2 is complete portability, which is because bytecode transformation technique is used for both CPU time and memory resource controlling, instead of modified Java run time systems. Before being loaded by the JVM, bytecode is modified in order to account for resources. For the memory resource, before every memory allocation instruction, code for accounting is inserted. CPU time accounting in JSeal2 is based on measuring the number of executed bytecode instructions, so code for CPU accounting is inserted to every basic block of code.

2.3.1.3 NOMADS

NOMADS [38] is a mobile agent system which has the ability to control the resources consumed by each agent. The NOMADS execution environment is based on a special virtual machine, which is called the *Aroma Virtual Machine* [37]. For every agent in the NOMADS system, there is an underlying Aroma VM running. Aroma VM is designed for capturing execution state of threads. It is implemented using C++ and includes the VM library and the native code library. The VM library can be linked to other application programs, and the native code library implements native methods in the Java API and it will be loaded by the VM library automatically.

Using a special virtual machine to control resource consumption has both advantages and disadvantages. Having a special virtual machine can support strong mobility of mobile agents, even when multiple concurrent threads coexist together, and it also provides a way to control resource dynamically on a fine-grained level. But it is quite difficult to design a new VM with both compatibility and good performance. Furthermore, in NOMADS, one instance of Aroma VM for each agent causes a very high overhead.

2.3.1.4 Java Resource Management API

Java Resource Management API [7] was proposed to be a widely applicable resource management interface for Java platform. It was recently developed by Java in collaboration with JSeal2's developers, in order to extend resource management support in Java.

The unit of resource management in Java Resource Management API is an *Isolate*, which is an encapsulation of a Java program. Isolates do not share state with each other. Resources in the RM API are represented by a set of resource attributes. A *dispenser* isolate is responsible for monitoring available resources and it serves as the connection between the resource implementation and the RM API. Resource consuming policies are encapsulated by *resource domains*, which may specify the reservations of resources and actions that should be executed upon certain events.

Implementation of RM API has been prototyped on top of the `Isolate` API. One of the most important features of RM API is the wide applicability. Besides the traditional resource such as CPU processor time, heap memory, RM API makes it possible for users to control their own resource by defining the resource attributes.

Java RM API is an extension developed by Java, so the code is portable across Java implementations. It is different from JSeal2 and JRes, which modify Java bytecode.

2.3.2 Resource Management Models

In this section, two theoretical models for resource management are reviewed: *Quantum* [34] and CyberOrgs [19]. Both of these models have hierarchical structures. However, there are significant differences in how resource delivery is modeled. Additionally, CyberOrgs model the notion of eCash separately from resources, enabling dynamic pricing, which is not supported in the Quantum model.

2.3.2.1 Quantum

Quantum, proposed by Luc Moreau and Christian Queinnec, is a theoretical model for resource management.

The semantics of Quantum was proposed in 1997 and later it was extended [35] in order to handle distributed and multi-type resources. In Quantum, the resource that computations need to execute is represented by *energy*. The basic resource control unit in Quantum is called a *group*. A group hosts a set of computations, and it also serves as a tank of energy.

In the Quantum model, a group can create new groups, so a hierarchy structure is generated. Each new group is assigned an amount of energy when it is created, and the energy is used for sponsoring the computations in this group. Computations consume energy from the sponsoring group, and if a computation needs more energy than what is available in the group, an energy `exhaustion` primitive is invoked to signal that the current group has run out of energy; if all the computations complete in one group which does not sponsor any sub-groups, the event group `termination` is signaled, and all the remaining energy is returned to the parent group.

Group creation, energy exhaustion, and group termination allow flow of energy between a group and its sub-groups, but energy may also flow between groups independently of the group hierarchy using another two primitives: `pause` and `awake`. `Pause` forces a group and all its sub-groups to be exhausted, and all the energy in this whole hierarchy is transferred to the group which called and sponsored the `pause` operation. Similarly, a group may also transfer energy to an exhausted group in order to make it awake, the group which calls `awake` sponsors the execution of the `awake` primitive.

2.3.2.2 CyberOrgs

CyberOrgs is a model for hierarchical coordination of resource usage by multi-agent applications in a network of peer-owned resources. Each cyberorg encapsulates a set of computations which are executed concurrently, and an amount of resource. A

concurrent computation consumes resource, which is allocated to it by its containing cyberorg. A cyberorg has a contractual relationship with its containing cyberorg, and it may purchase resources from its containing cyberorg according to the signed *contract*. The currency that flows among cyberorgs is called eCash.

CyberOrgs organizes resources and computations as a tree. Each cyberorg except the root cyberorg is contained inside another cyberorg. Figure 2.3 shows the hierarchical structure of a cyberorg. Black dots represent computations and ellipses represent cyberorgs. A cyberorg hosted by another cyberorg purchases resources it needs from the host cyberorg, according to a pre-negotiated contract. This contract, which must be signed between two cyberorgs before one is hosted by the other, stipulates the types and quantities of resources which will be available to the hosted cyberorg as well as their costs. After satisfying its contractual obligations, a cyberorg distributes the remaining resources available to it among the computations it is managing according to its own local resource distribution strategy.

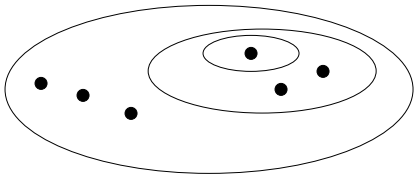

Figure 2.3: Hierarchical Structure of a Cyberorg

CyberOrgs distribute resources through several primitives. In this section, the resource control primitives in CyberOrgs model are reviewed.

- `Isolate`:

  As shown in Figure 2.4 (a), one cyberorg may create another cyberorg inside it using the `isolate` primitive. A number of actors (computations), messages, and some eCash are encapsulated by the new local client cyberorg. There is a contract between the new cyberorg and its host cyberorg, which is used to determine the trade of resources.

- `Assimilate`:

  As shown in Figure 2.4 (b), a local cyberorg is assimilated inside its host cy-

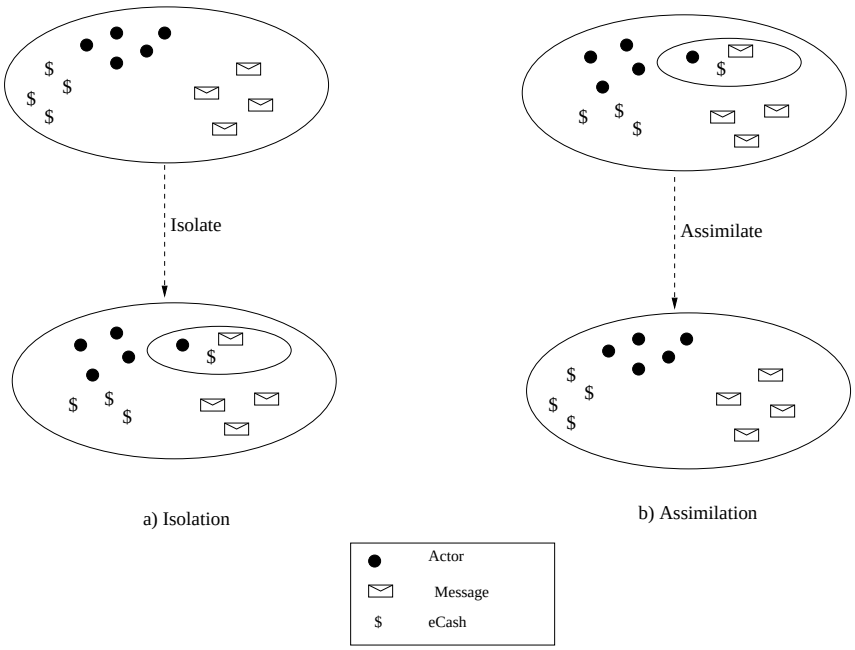

Figure 2.4: CyberOrgs Primitive Operations: Isolate & Assimilate

berorg through the `assimilate` primitive. All the contents of the assimilating cyberorg (actors, eCash, and messages) become contents of the host cyberorg after the assimilation. Furthermore, the contract between the assimilating cyberorg and its host ceases to exist.

- `Migrate`:

  A cyberorg may realize that its resource requirement has exceeded what is offered by its contract with the host cyberorg. This triggers its attempts to migrate, as shown in Figure 2.5. A cyberorg may migrate from one host cyberorg to another. However, this must be preceded by negotiation of the terms under which the client may be hosted. The tasks required for a cyberorg to migrate are as follows:

  1. Search:

     Before migration, the cyberorg searches for a potential host which can provide sufficient resources.

  2. Negotiate:

     After searching, the migrating cyberorg negotiates with the potential host

in order to sign a new contract with it.

3. Migrate:

   Once the negotiation succeeds and a new contract is agreed, the cyberorg can migrate to the selected host, receive an amount of resources, and pay an amount of eCash according to the contract.

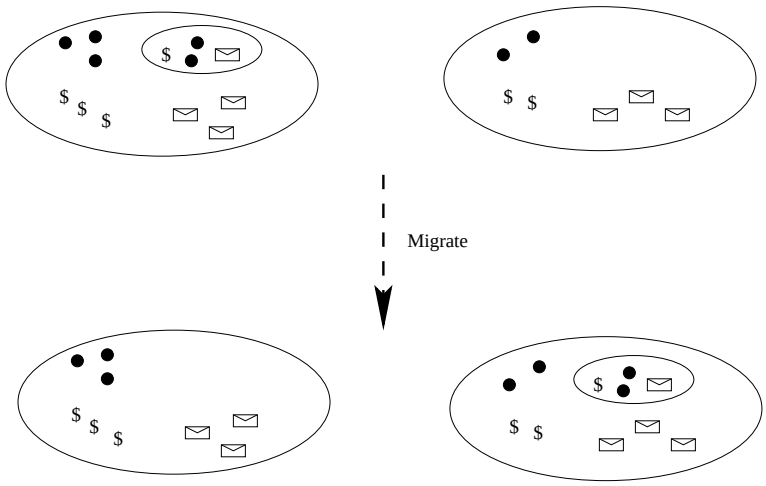

Figure 2.5: CyberOrgs Primitive Operations: Migrate

In the CyberOrgs model, resources can be controlled using the above primitives. Resources flow from the root to the leaves of the cyberorgs hierarchy, and the eCash flows in the opposite direction. Through migration, a cyberorg can obtain resources from any cyberorg as long as a contract has been negotiated.

2.4 CPU Resource Allocation

We focus on CPU time resource as the resource to be controlled in our CyberOrgs implementation, and the most important component of a system for controlling CPU time is the scheduler. Existing research work on scheduling CPU resource is reviewed in this section.

CPU resource allocation is extensively studied and many approaches have been proposed in the form of scheduling algorithms or scheduling schemes.

A scheduling algorithm specifies a set of rules that determine which process is to be executed at any particular moment. Some widely used scheduling algorithms are reviewed as follows:

- *Round Robin Scheduling* [39]: The round-robin scheduling is a scheduling algorithm which prevents starvation. In round-robin scheduling, every process is put in a queue and given the same amount of CPU time to be executed until it is finished. Once a process completes, it is removed from the scheduler's queue.

- *Rate Monotonic Scheduling (RMS)* [27]: The rate-monotonic scheduling algorithm assigns fixed priorities to processes . It is a preemptive and priority driven algorithm, which means that whenever there is a request for a task with higher priority than the one currently being executing, the executing one will be interrupted, and the newly requested task with higher priority will be executed instead. In *RMS*, the priority assigned to a task has a monotonic relation to the request rate of that task, which means the shorter the period of a task, the higher its priority.

- *Earliest Deadline First Scheduling (EDF)* [29]: The earliest-deadline-first scheduling is a dynamic scheduling algorithm. All processes are maintained in a priority queue. At the end of time-slice of a process, it is put at the end of the queue, the queue will be searched for the process which is closest to its deadline, and this process will be scheduled next. By using the dynamic priority, the performance of earliest-deadline-first scheduling is better than rate-monotonic scheduling which uses the static priority for each process.

A scheduling scheme is a way of combining different scheduling algorithms for a set of applications with different constraints for CPU time. For instance, in CPU inheritance scheduling [11], scheduling based on EDF is combined with multi-priority-based round robin algorithm for scheduling a mix of real time and interactive applications. In a paper by Manoj Lal and Raju Pandey [26], a CPU resource allocation

scheduling scheme is presented, which combines three scheduling algorithms for scheduling both real time and non-real time mobile programs.

2.5 Chapter Summary

Work in several related areas was reviewed in this chapter, including models of concurrency, resource management approaches, as well as CPU resource allocation algorithms. The Actor model is one of the formal models for concurrent computation. CyberOrgs is a theoretical model for resource management, which organizes resources and multi-agents' computations in a hierarchy. Several scheduling schemes for CPU resources were reviewed in this chapter.

Chapter 3

CyberOrgs Implementation

3.1 Introduction

An implementation of CyberOrgs has been developed by extending Actor Architecture [25], which is a Java library and run-time system for supporting actors. In our CyberOrgs implementation, each actor needs processor time resource to carry out its computation, and the resource is distributed to each individual actor by the cyberorg which hosts it according to a local resource allocation policy. Furthermore, the hierarchical scheduling for processor time in CyberOrgs has been implemented scalably by converting the hierarchical schedule into an equivalent flat schedule on the fly. Implementation details are discussed in this chapter, including the Actor Architecture framework, CyberOrgs system design, and our scheduling scheme as well as its performance analysis. The flat scheduling scheme can be generalized for distributed processor time control, and the implementation of distributed CyberOrgs as well as its overhead analysis are presented. The chapter ends with description of specific scenarios of using CyberOrgs.

3.2 Actor Architecture

Actor Architecture (AA) is a middleware system architecture that provides an execution environment for actors. An instance of the AA run-time system is called a *platform*. Each platform has four layers with eight components as shown in Fig-

ure 3.1.

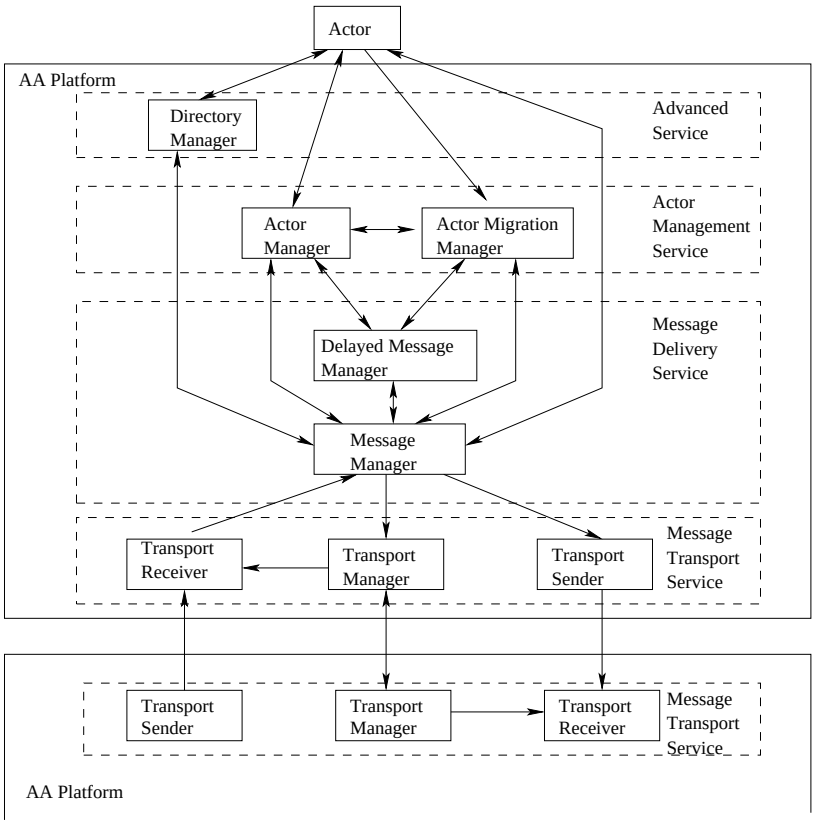

Figure 3.1: AA Platform Structure ([22])

- *Message Manager (MM)*:

  MM handles all messages in a AA platform. Every message passes through at least one message manager. If the destination actor of a message is on the current AA platform, the MM delivers the message directly, otherwise it delivers the message to the MM on the AA platform where the destination actor is located.

- *Transport Manager (TM)*:

  TM forms the central part of the message transport service layer, which is an AA platform's interface with other AA platforms.

- *Transport Sender (TS)*:

  TS is responsible for receiving messages from the MM of the current AA plat-

form and sending each of these messages to the Transport Receiver (described below) of the destination AA platform, provided there is a connection between these two AA platforms. If there is no connection, TS communicates with the TM on the destination platform to open a connection for message delivery.

- *Transport Receiver (TR)*:
  TR delivers the messages it receives to the destination actors on the current AA platform.

- *Delayed Message Manager (DMM)*:
  Actors may migrate at any time, causing delivery of messages intended for such actors to be delayed. DMM is responsible for holding these messages temporarily, and delivering them when it is possible.

- *Actor Manager (AM)*:
  AM manages states of all mobile agents on the AA platform.

- *Actor Migration Manager (AMM)*:
  AMM manages all migrations that happen on the AA platform.

- *Directory Manager (DM)*:
  DM provides advanced services, such as matchmaking and brokering.

3.3 CyberOrgs System Design

Actor Architecture provides an execution environment for actors and it supports actor primitives, such as sending/receiving messages, creating new actors, and changing local state. In Actor Architecture, resource allocation relies on the underlying Java Virtual Machine (JVM). We extend AA to support CyberOrgs, making the resource control visible to programmers.

Figure 3.2 shows the design of the CyberOrgs implementation. We extend Actor Architecture by adding two key components: *CyberOrg Manager* and *Scheduler Manager*. CyberOrg Manager is the central component of each CyberOrgs platform.

All resource control operations are carried out by the CyberOrg Manager. The results of such operations are sent to Scheduler Manager, which schedules all actors in the platform according to these results.

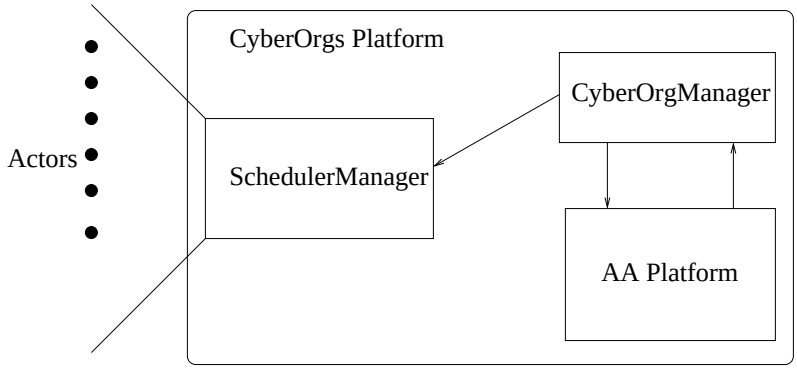

Figure 3.2: CyberOrgs System Design

Two service layers, *CyberOrg Service* and *Scheduler Service*, are built on top of the original AA platform to extend it to a CyberOrgs platform. The detailed structure of the CyberOrgs platform is shown in Figure 3.3. Each actor in the platform is encapsulated by some cyberorg. Resource acquisition and control is achieved by programmers using CyberOrgs primitives. The features and functions of individual components in a CyberOrgs platform are described in the following sections.

3.3.1 Cyberorgs

A cyberorg is the basic unit of resource acquisition and control in a CyberOrgs platform. Actors, messages, and eCash are encapsulated in a cyberorg. Cyberorgs are organized as a hierarchy in the platform. Each cyberorg holds a list of actors, some units of eCash, an amount of resource, and a list of client cyberorgs. The contents of a cyberorg are described as follows.

3.3.1.1 Actors

Actors in a CyberOrgs platform represent concurrent computations. An actor carries out computational tasks and requires resources to perform its tasks. Each actor is

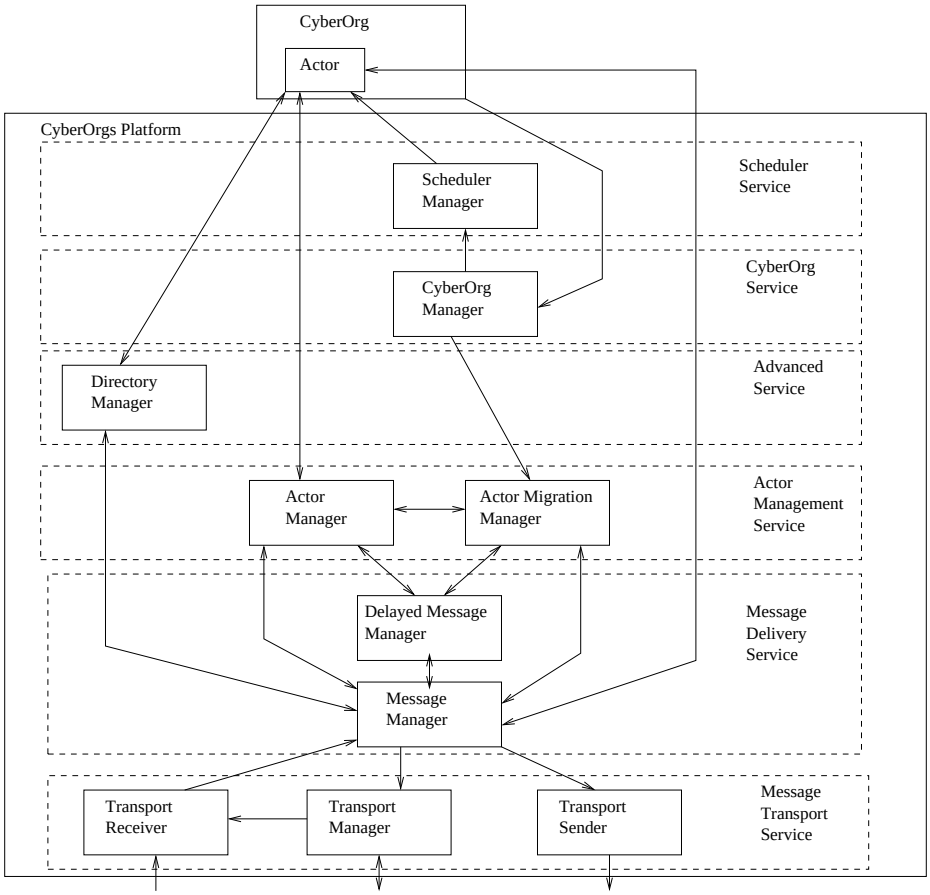

Figure 3.3: CyberOrgs Platform Structure

contained in a cyberorg and obtains resources from it.

3.3.1.2 Facilitator

Facilitator is a special actor which facilitates the execution of the *application actors* (which carry out functional computations) in a cyberorg. It receives an amount of resource from its host cyberorg to support its actions just as the application actors do, but it is different from application actors, because the messages it processes are requests for CyberOrgs primitives. Every request for a primitive cyberorg operation eventually arrives at the corresponding facilitator of the specific cyberorg as a message. While processing the message, the facilitator invokes the requested primitive of the host cyberorg.

3.3.1.3 eCash

CyberOrgs trade in a market of resources, and eCash is the currency in this market. Each cyberorg holds certain amount of eCash, and the cyberorg uses its eCash to buy resources from its host cyberorg in order to support its computations.

3.3.1.4 Resource

In our CyberOrgs implementation, we only deal with processor time resource. Initially all resources in the system belong to the root cyberorg, and other cyberorgs can purchase resources from their respective host cyberorgs over the course of time.

3.3.1.5 Client Cyberorgs

Cyberorgs are organized as a hierarchy in the system. Each cyberorg maintains a list of client cyberorgs. There is a contract between a host cyberorg and each of its client cyberorgs. For each client, this contract specifies the amount of resource that the client cyberorg can obtain from the host cyberorg, as well as the amount of eCash it would pay for the resource. According to the contract, the client cyberorg purchases resource in order to support the computations which are carried out by its actors.

3.3.2 CyberOrgs Primitives

The primitive operations of CyberOrgs are implemented as methods of the `CyberOrg` class. These methods are called by the facilitator in a cyberorg. Through the primitives, a cyberorg may change its own state. For example, when the `Isolate` primitive is invoked, the cyberorg modifies its list of actors, and the list of client cyberorgs according to the parameters of the isolation.

3.3.3 CyberOrg Manager

An important component in a CyberOrgs platform is the CyberOrg Manager. Most computations about resource acquisition and control are carried out by the CyberOrg

Manager, and this is where the CyberOrg primitives are eventually carried out. It interacts with the *Scheduler Manager* (described in Section 3.3.4) in order to implement processor time control. In this section, we describe the functions of CyberOrg Manager in detail.

3.3.3.1 Maintaining Cyberorgs Structure

The CyberOrg Manager maintains the hierarchical structure of the cyberorgs on a CyberOrgs platform. When a CyberOrgs primitive is invoked, the CyberOrg Manager is responsible for modifying the hierarchy to reflect the change in the cyberorgs structure. CyberOrg Manager is the only component on the platform which knows the global structure of the cyberorgs hierarchy.

3.3.3.2 Handling CyberOrgs Primitives

The CyberOrg Manager handles all of the CyberOrgs primitives that are invoked in a platform. The requests for CyberOrgs primitives come from the facilitators which represent corresponding cyberorgs. According to the parameters of a request, the CyberOrg Manager computes the amount of resource for each actor that is involved, and coordinates with the Scheduler Manager in order to allocate CPU resource based on the results.

- Isolate:

  When `isolate` is invoked in a cyberorg, a new client cyberorg is created in it, and some of its actors are isolated to the new client. The CyberOrg Manager is responsible for calculating the updated amounts of resource which are to be given to these actors according to the new contract between the client cyberorg and the host cyberorg. Consequently, some fields in the host cyberorg are changed and need to be updated, such as the eCash, the list of client cyberorgs, and so on.

- Assimilate:

  A client cyberorg disappears as a result of the `assimilate` primitive, and all

of its contents are released to its host cyberorg. According to the local policy of the host cyberorg, the CyberOrg Manager calculates how much resource to give to the actors of the assimilating cyberorg. In the meantime, because there are contents which are coming into the host cyberorg, the corresponding fields have to be modified.

- Migrate:

  There are two types of migration in the CyberOrgs platform: intra-platform migration and inter-platform migration. We use different ways to deal with them.

  - For the intra-platform migration, a cyberorg migrates from its original host cyberorg to a new host cyberorg (with which a contract has already been reached by negotiation) on the same platform as the old host. In this case, no objects need to be moved across machines. What the CyberOrg Manager needs to do is similar with what is done in isolation and assimilation: it recalculates the CPU resource for the migrating actors according to the new contract, and modifies the state for both the original and the new host cyberorg.
  - For the inter-platform migration, a cyberorg migrates from its original host cyberorg to a new host cyberorg which is located on a different CyberOrgs platform. In this case, the calculation and modification are similar with what is described in the intra-platform migration. Besides, actors which are hosted in the migrating cyberorg have to be migrated across machines, and this can be done by interacting with the *Actor Migration Manager* component of the AA platform. Furthermore, information related to the state of the migrating cyberorg is sent to the destination platform as a message. The CyberOrg Manager in the destination platform takes this message, extracts the state information from it, and creates a cyberorg locally with the same state as shown in the message. This newly created cyberorg is put into the local Scheduler

Manager in order to be assigned an amount of resource according to the contract that the migrating cyberorg signed with the new host cyberorg. After the creation of the new cyberorg on the destination platform, the old cyberorg which invoked the migration is destroyed on its platform.

3.3.3.3 Interacting with Scheduler Manager

After computing resource allocations, the CyberOrg Manager interacts with the local Scheduler Manager which enforces processor time allocations on the actors. Each primitive operation changes the resource allocation, and it is the CyberOrg Manager that translates these changes for the Scheduler Manager, which eventually makes changes in the schedule.

3.3.4 Scheduler Manager

Scheduler Manager is a round-robin-like thread scheduler which uses Java's `suspend` and `resume` primitives to schedule actor threads in a flat queue. Each actor is scheduled for an amount of time which is calculated and updated for each thread dynamically by the CyberOrg Manager.

An important objective of our scheduling scheme is to decrease the overhead of scheduling. A hierarchical implementation of the scheduler would result in very high overhead, as shown by the analysis in [19]. Based on the concern of efficiency, the Scheduler Manager does not perform any computations other than the scheduling of threads. Specifically, the Scheduler Manager should not be aware of cyberorgs structure, and it should only deal with actor threads.

In our implementation, the Scheduler Manager module is responsible for scheduling all actor threads in the system in a round-robin fashion. It uses a queue to maintain the actor threads to be scheduled, and it also has a hash table to match each thread to a certain amount of CPU time which is allocated to it. The main body of a Scheduler Manager is an infinite loop. It gets a thread from the head of queue, schedules it for certain amount of time, and then puts it back at the tail of

the thread queue (if it is still alive). The algorithm of the scheduler is shown in Figure 3.4.

```
1.  begin schedule

2.   while (true)
3.    if the Queue is not empty;
4.     get the first thread in the Queue;
5.     get the certain time slice for the thread from hash table;
6.     schedule the thread for time slice;
7.     if the thread is alive;
8.      put it to the tail of Queue;
9.     go to 3;
10.    else sleep for some time
11.   end while;

12.  end schedule
```

Figure 3.4: Algorithm of Scheduler Manager in CyberOrgs System

Alternatively, there is another approach which uses thread priority instead of `suspend/resume` to schedule actor threads. The thread for the Scheduler Manager itself has the highest priority in the system, and actor threads have relative low priorities. Whenever an actor thread is to be scheduled, the scheduler assigns a higher priority to the actor thread than other actors and goes to sleep, so that the actor thread with higher priority can get a chance to execute. After certain amount of time, the Scheduler Manager wakes up, decreases the priority of this thread, and schedules the next thread in the queue.

Experiments based on both approaches have been carried out and the results are presented in Chapter 5. However, because we use Java to implement the whole system, and the thread priority enforcement in Java is not precisely specified, the `suspend/resume` approach is more reliable. Although `suspend` and `resume` are deprecated in Java in order to avoid deadlock, they are safe to use on actors because multiple threads do not access the same object.

3.4 Scheduling CyberOrgs

In the previous sections we have described the CyberOrgs system design. One important concern is the scheduling scheme. Note that we only have one scheduler for each platform, instead of a hierarchy of schedulers conforming to the structure of cyberorgs. In this section, we discuss our flat scheduling approach [21] in detail, and present experimental results which show that the approach is efficient.

The hierarchical structure of CyberOrgs poses a challenge for efficient scheduling of cyberorgs. Intuitively, to implement a hierarchy, we can use a tree of schedulers which have exactly the same structure as the cyberorgs. Figure 3.5(a) shows a cyberorg structure, and Figure 3.5(b) shows a hierarchical scheduler for scheduling the cyberorgs. Every cyberorg in the hierarchy has its own scheduler (represented by the circle), which is responsible for scheduling all actors (represented by the black dots) as well as the schedulers of for its client cyberorgs. In the hierarchical implementation, the number of scheduler threads is the same as the number of cyberorgs. In order to switch from one scheduler thread to another, we have to use `suspend/resume` primitives which are very costly. Furthermore, whenever one scheduler thread is suspended/resumed, all threads which are scheduled under it have to be suspended/resumed down to the leaf. For instance, in Figure 3.5(b), if scheduler S_b is suspended, scheduler S_c and actor thread c_1 are also going to be suspended.

An overhead analysis has been carried out for the multi-layer scheduler scheme in [19]. Suppose o is the overhead resulting from a single `suspend/resume`. Let us consider the case that every scheduler in the hierarchy schedules a leaf thread or a scheduler thread under it for a fixed time slice t. A scheduler has to suspend all running threads down to the leaf in order to suspend a thread under it. In any time period t, there would be h suspend overheads for a scheduler at each height h, amounting to $(h^2 - h) \times o/2$ overheads, and the overhead resulting from resuming a thread would be similar. Therefore, the overhead caused by `suspend/resume` increases when the height of the scheduler hierarchy increases, and it could be pro-

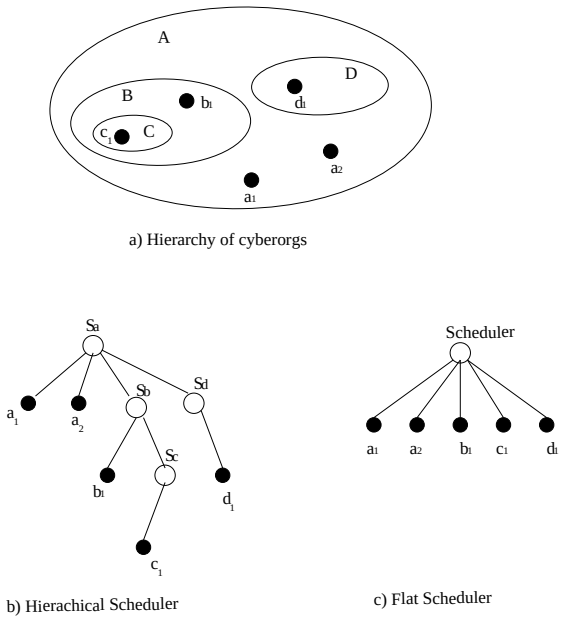

Figure 3.5: Comparison of Hierarchical Scheduler and Flat Scheduler of CyberOrgs Implementation

hibitively heavy if h is large. As a result, enforcing a hierarchical coordination is very costly [30].

A contribution of our implementation is in developing a mechanism for efficiently converting a hierarchical scheduler to an equivalent flat one , and then enforcing it to schedule all threads according to the same resource allocations as required by the hierarchical schedule but without the heavy overhead.

The flat scheduler is shown in Figure 3.5(c). There is only one global scheduler for the whole platform, and it is responsible for scheduling all threads in the platform. The amount of resource allocated to each thread is calculated according to the original hierarchical structure. This is possible because initially all available resources are possessed by the root cyberorg, and all the actor threads hosted by it as well as the client cyberorgs receive part of its resources, which can be represented by a function of the amount of total available resource. Consequently, for every actor thread, we can calculate the amount of resources allocated to it using its cyberorg's resources and the local distribution policy. Each cyberorg's allocation is similarly a known part of its host's resources. After this calculation, all computation threads

can be scheduled directly by the global scheduler and multiple schedulers are no longer necessary. Compared to the hierarchical scheduling, creating a flat scheduler is efficient because there is only one scheduler thread for the whole CyberOrgs platform. The overhead analysis on our scheduling scheme is described in the next section.

3.4.1 Overhead Analysis

There are two types of overhead in our flat scheduler. First, the round-robin scheduling itself causes overhead when it is attempting to switch from one thread to another. Apparently, this type of overhead is related to the number of threads in the system, and the time slice of each thread being scheduled. Second, additional overhead results from the changes to be made on the schedule. When a CyberOrgs primitive is invoked, the scheduler needs to be modified because there are certain changes on the resource allocation. The overhead caused by the modifications is related to the rate at which CyberOrgs primitives are called. However, this type of overhead is also manageable, because the data related to resource allocation is maintained in a hash table, and the number of changes in this hash table as a result of a primitive operation is linearly related to the number of local actors involved in the primitive. Compared with the first type of overhead, the second one is less frequently caused, because the first type of overhead results from every switch in the entire schedule, whereas the CyberOrgs primitives are invoked only when it is necessary to change the resource allocation.

Experiments have been carried out to figure out the relationship between overhead and a number of factors, including the number of threads, the structure of cyberorgs, and the size of the time slice.

3.4.1.1 Overhead Vs. Number of Threads

For a flat scheduler, the main overhead comes from the schedule construction and the switching from one thread to another. So the overhead increases when the length

of thread queue increases. Assuming o is the overhead, n is the number of threads in the system, then

$$o \sim n$$

We carried out our experiments in the following way:

- Create a number (n) of threads which carry out identical simple computational tasks.
- Run the threads allowing Java thread scheduler to schedule them, and calculate the time t_1 it takes to finish all threads.
- Run the same threads in our scheduler, and invoke `isolate` primitives on a random number of threads to create a hierarchy of cyberorgs, and calculate the time t_2 in which all the threads complete.
- Calculate the additional overhead of scheduling n threads organized in a hierarchy of cyberorgs, which is $t_2 - t_1$

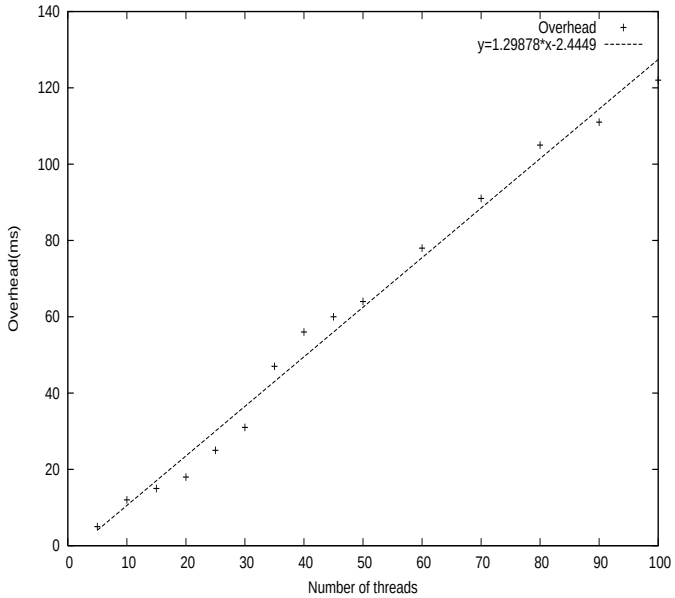

Figure 3.6: Overhead of the Flat Scheduler (number of threads in the system: 5-100)

Figure 3.6 and Figure 3.7 show the results of our experiments. Figure 3.6 displays 15 samples, and each of them represents the overhead of scheduling certain number

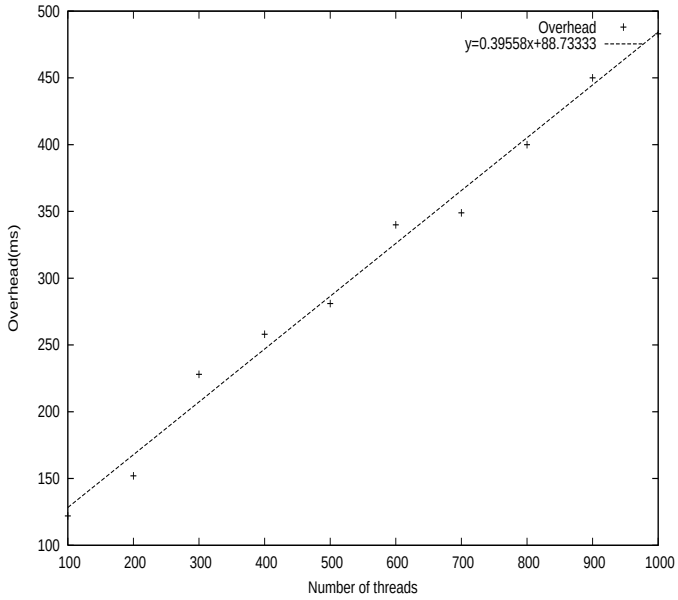

Figure 3.7: Overhead of the Flat Scheduler (number of threads in the system: 100-1000)

of threads (the number is between 5 and 100) using our flat scheduler. Similarly,in Figure 3.7, 10 samples are shown, and the data for these 10 samples are collected from the scheduler when it schedules from 100 to 1000 threads. The value of each point in the figures is the average of five repeated experiments to guarantee the accuracy of the results, because there are many other factors that will influence the calculation time such as the stability of the computer, background workload, and so on.

Least-square linear regression [18] is used to fit the sample points by straight lines as shown in the two figures. The correlation coefficients (R) are 0.99433 and 0.99464 respectively, and the standard deviations of the regressions are 4.265 and 13.2 respectively, which indicate that the regressions are are acceptable. Therefore, the overhead of our scheduler has a linear relation with the number of threads in the system.

Another interesting point to note is: we notice that the slopes of the two lines are different. In Figure 3.7, which ranges from 100 to 1000 threads, the slope is much less than that of Figure 3.6.

3.4.1.2 Overhead Vs. CyberOrgs Structure

The previous experiments showed that the overhead increases when the number of threads increases, but does the structure of cyberorgs influence the overhead? Experiments were carried out to studying the relationship between the overhead and the cyberorgs structure when using the flat scheduler.

Three types of extreme structures of cyberorgs were investigated in these experiments: one layer, wide tree, and deep tree.

One Layer: In Figure 3.8, all threads are held by one cyberorg, which does not have any client cyberorg. This structure has only one layer.

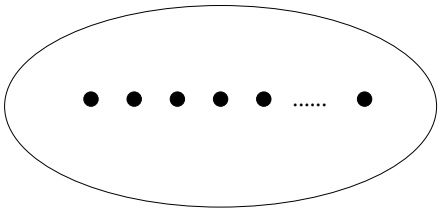

Figure 3.8: CyberOrgs Structure I: One Layer

Wide Tree: In Figure 3.9, the root cyberorg has a number of client cyberorgs which do not have any client cyberorgs of their own, and the whole structure is like a very wide tree.

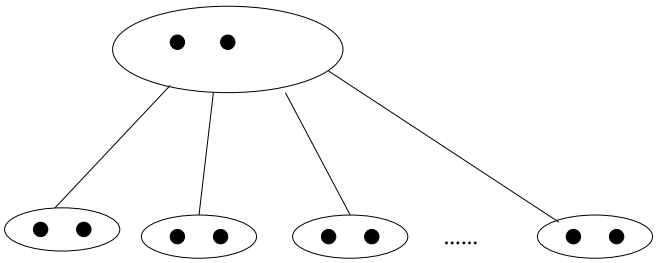

Figure 3.9: CyberOrgs Structure II: Wide Tree

Deep Tree: In Figure 3.10, each cyberorg has only one client cyberorg except the one on the leaf. This structure represents a very deep tree.

In the following experiments, we assign 10 milliseconds of time slice to each thread in one scheduling cycle, and schedule them in the above three kinds of structures respectively. The results are shown in Figure 3.11. Each value of the points

in the figure is the average of five identical experiments in order to minimize the influences from external factors that may exist.

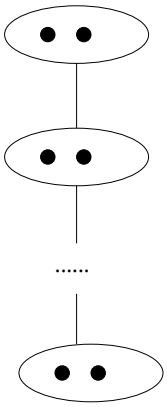

Figure 3.10: CyberOrgs Structure III: Deep Tree

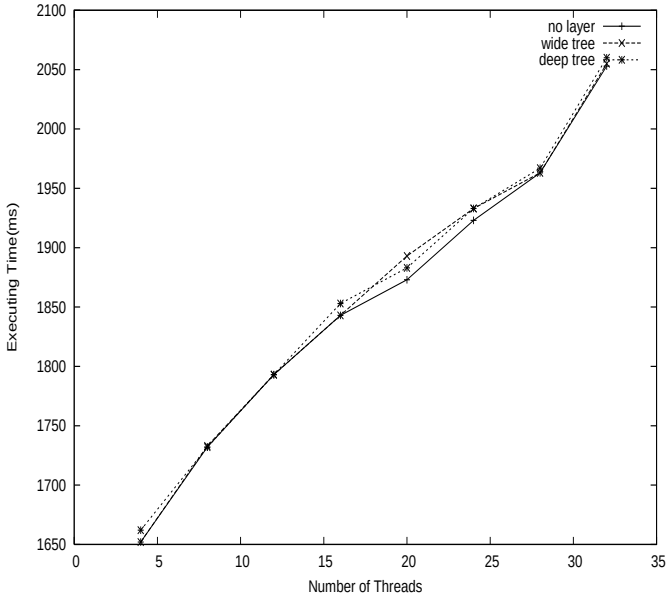

Figure 3.11: Overhead Comparison for Different CyberOrgs Structures (No Layer, Wide Tree, and Deep Tree)

CyberOrgs Structures	No Layer	Wide Tree	Deep Tree
Slope	13.04	13.21	13.12
Intercept	1619.25	1620.22	1624.07
Standard Deviation	17.50	17.32	16.42

Table 3.1: Linear Regression for the Overhead of Different CyberOrgs Structures (No Layer, Wide Tree, and Deep Tree)

Table 3.1 shows the parameters of linear regressions for the overhead from different types of cyberorgs structures. We conclude that the overhead in the three cyberorg structures has no significant difference, because the regressions illustrate that the sample points from the three experiments are close to similar straight lines. It means that the height or width of the tree structure of cyberorgs has insignificant influence on the overhead. [1]

3.4.1.3 Overhead Vs. Smallest Time Slice

Figure 3.12 shows the relationship between the overhead and the smallest time slice for which a thread may be scheduled [2]. In the experiments, we scheduled 100 threads which carried out identical computations in our scheduler, and varied the time slice from 3 to 40.

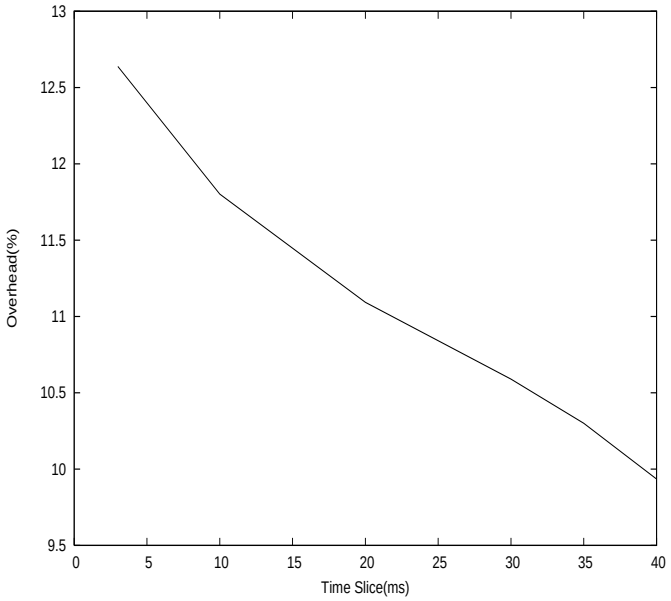

Figure 3.12: Overhead Vs. Time Slice: Overhead can be controlled by increasing the minimum time slice that is allowed in the system

In Figure 3.12, it is shown that increasing the smallest time slice can reduce the overhead of the scheduler, because when the smallest time slice is increased,

[1] We use these three extreme cases to test our scheduler, but cyberorg structure does not have to be any one of them. Cyberorgs may construct any type of tree.

[2] All threads are not scheduled for identical time slices in a cyberorg system.

the number of switches between threads in a fixed time period is reduced. Some opportunities for managing the overhead arising from this observation are examined in the next section.

3.4.2 Overhead Control

In the previous section, we have analyzed the overhead of the flat scheduler, and it turns out to be linear with respect number of switches between threads. This fact could be used to control overhead in the system.

3.4.2.1 Overhead Control by Time Slice Constrains

Considering we schedule a number of threads in a time cycle, say T, and each thread $thread_i$ will get a certain time slice t_i to execute in T. Let o be the overhead that results from switching from one thread to another. Because the majority of the overhead of a flat schedule comes from switching between threads, reducing the number of these switches can reduce the overhead. In order to control the overhead within an acceptable range (e.g. 3%), we can constrain the time slice and the time cycle. We present some examples of constraint calculation according to the distribution of time slice.

- Constant Time Slice:

  In this case, $t_i = t$. So in each time cycle of T, the number of switches is: $\frac{T}{t}$. If we want the overall overhead to be within p percent of T, we should have: $(\frac{T}{t}) \times o < pT$. The constraint for constant time slice is:

  $$\frac{o}{t} < p$$

  which can be satisfied by ensuring: $t > \frac{o}{p}$

- Binary Time Slice:

  In this case, when we add a new thread to the schedule, it gets half of the remaining time in T. That is: $t_i = (\frac{1}{2})^i \times T$. Let n be the number of threads

that could be handled in T, then we have $n \times o < p \times T$, i.e. $n < \frac{pT}{o}$. So the inequality for time slice would be

$$t_i > (\frac{1}{2})^{(\frac{pT}{o})} \times T$$

- Some Percentage of Remaining time in T:

  We extend the previous distribution to a common case; that is, every added thread gets a certain percentage x of remaining time, i.e., $t_i = x \times (1 - \sum_{j=1}^{i-1} t_j) \times T$. Because the number of switching should be kept within $\frac{pT}{o}$ as the previous case, the inequality for time slice would be

$$t_i > x \times (1 - \sum_{j=1}^{\frac{pT}{o}-1} t_j) \times T$$

- Random Time Slice:

  In this case, t_i is a random integer number. This is the most common one. We can guarantee that the percentage of overhead is less than p by keeping every random time slice $t_i > \frac{o}{p}$ as in the first case. Although it is sufficient, it is not a necessary condition.

3.4.2.2 Overhead Control by Adjusting Granularity

Another opportunity for overhead control is by adjusting the granularity of control. We set two parameters for the scheduler: *SmallestTimeSlice* and *LargestTimeSlice* to help manage overhead.

The SmallestTimeSlice specifies the smallest time slice that the CyberOrgs system is allowed to accept. On the contrary, the LargestTimeSlice puts a limit on how large a time slice could be. These two parameters together constrain all time slices in the system.

As for requests which are asking for time slices smaller than the SmallestTimeSlice, they are not discarded right away. When such a request comes, we scale up

all of the time slices in the system (including the small request) in order to make the request acceptable. This strategy makes the resource control more coarse, but time slice requests that are smaller than SmallestTimeSlice can be entertained in the system.

If we do not put a limit on the scale-up, the resource control will become increasingly coarse. Therefore, we have the LargestTimeSlice which eventually limits the total amount of scale-up. The LargestTimeSlice specifies the upper limit of a time slice in the system. In the scale-up process, the system keeps track of the largest time slice. If the largest time slice becomes larger than LargestTimeSlice, as the system is attempting to accept a request for a small time slice, it rejects the request.

3.5 Distributing CyberOrgs

On the basis of the single-node CyberOrgs system, a distributed version has been developed. Distributed CyberOrgs can be used to control resources which are located on computers distributed over a network. In this Section, we describe the design of our distributed CyberOrgs system.

3.5.1 Distributed Scheduler

A single-node CyberOrgs scheduler with flat structure has been described in Section 3.4, and it converts the hierarchical structure of the cyberorgs on a single node into a flat schedule for threads which is consistent with the resource allocation decisions made by cyberorgs in the hierarchy.

One challenge in implementing a distributed CyberOrgs system is developing a distributed scheduler, which is able to enforce a distributed schedule on a network of nodes.

Our design of the distributed scheduler is shown in Figure 3.13.

There are two characteristics of our distributed scheduler:

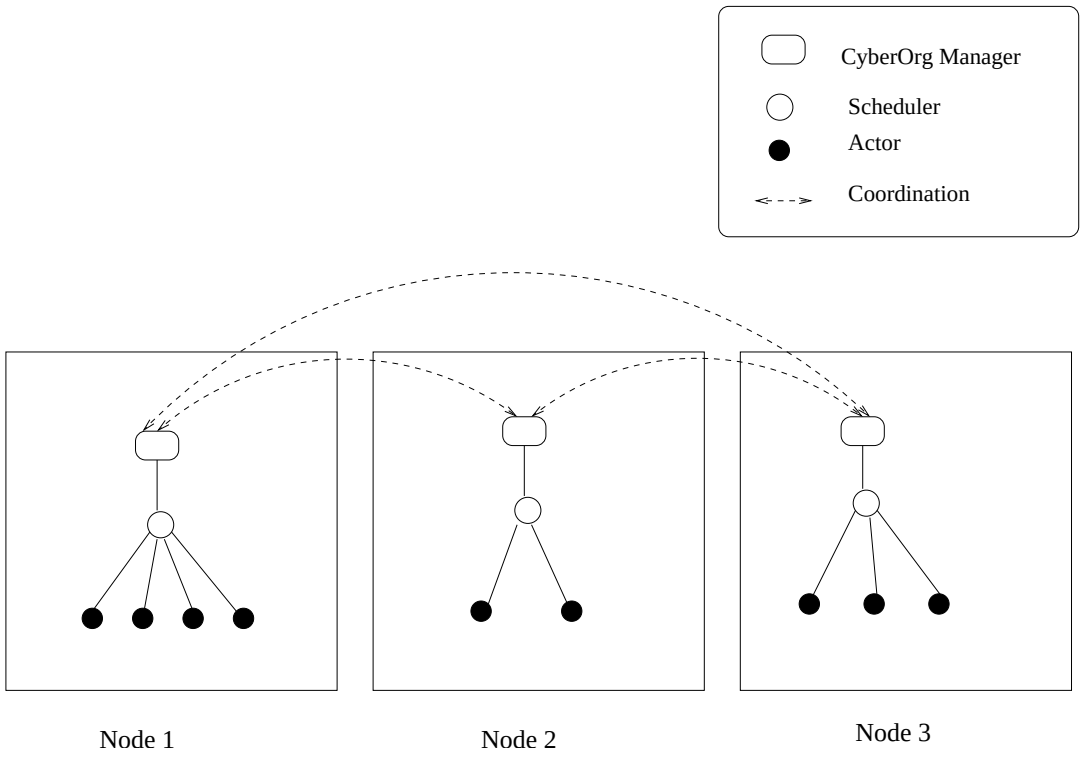

Figure 3.13: Structure of Distributed Scheduler in CyberOrgs System

- One sub-scheduler per node

  There is one sub-scheduler for each node of the CyberOrgs system. Every sub-scheduler has the flat structure (as described in Section 3.4); it schedules all actors on the local node on the fly and it is in charge of all the resources which are available to the corresponding node.

- Coordination between sub-schedulers

  The sub-schedulers in the distributed CyberOrgs system are located and manage resources on separate nodes, but they also cooperate with each other when necessary.

  As shown in Figure 3.13, the communication between two sub-schedulers has to pass through the CyberOrg Managers on their respective nodes. The CyberOrg Manager is an active component on any node: it receives messages from the environment, carries out calculations for resource allocation, and sends the results of calculations to the scheduler on the local node to invoke

the corresponding changes to resource allocation.

3.5.2 Distributed CyberOrgs

If each cyberorg must entirely locate on a single machine, we call the system a system of *distributed CyberOrgs*. In a distributed CyberOrgs system composed of multiple nodes, there is one CyberOrgs platform running on each individual node in order to deliver the local resources to the actors on that node. A cyberorg can migrate from one node to another in its entirety and is not allowed to be located separately on more than one node.

The components of a distributed CyberOrgs system are as follows.

- Cyberorg

  A cyberorg is created on one node and exists in its entirety on that node. It may migrate to another node after negotiation, but all of its contents have to be migrated together.

  A cyberorg has one and only one facilitator actor, and it also encapsulates application actors, some amount of eCash, and resources on the local node.

- CyberOrg Manager

  Every CyberOrgs platform needs a CyberOrg Manager, so in a distributed CyberOrgs system, there is one CyberOrg Manager on each node. These CyberOrg Managers are responsible for allocating resources on their own nodes and they communicate for inter-node coordinations. For example, when a cyberorg is about to migrate to another node, the local CyberOrg Manager contacts the CyberOrg Manager on the node that the cyberorg is migrating to, and sends the contract information to the remote CyberOrg Manager, which sets up resource delivery for the cyberorg with the Scheduler Manager on that node.

- Scheduler Manager

  We use the distributed scheduler which was described in Section 3.5.1. There is

one sub-scheduler per node, and these sub-schedulers communicate with each other though the CyberOrg Managers. A Scheduler Manager is responsible for scheduling all actors on the local node according to the predefined resource allocation policy. The allocation may be changed, and the changes are enforced by the CyberOrg Manager on the same node.

For the distributed CyberOrgs system, we analyze CyberOrgs primitives as follows:

1. Isolation:

   Because one cyberorg locates on one specific node in the distributed CyberOrgs system, isolation can be handled by the single node scheduler on that node.

2. Assimilation:

   Similar with isolation, assimilation is handled by the single node scheduler.

3. Migration:

   There are two types of migration in a distributed CyberOrgs system: intra-node migration and inter-node migration. In the former case, the new host of the migrating cyberorg is on the same node as the one from which it is migrating. The sub-scheduler on the current node does not need to interact with any other sub-schedulers, because there are no actors which are moved from one machine to another. In the latter case, the new host of the migrating cyberorg is on a different node than the one from which it is migrating. The sub-scheduler on the current node coordinates with the sub-scheduler on the destination node. After migration, the cyberorg receives resources from the new host according to the new contract between them, and its actors are scheduled by the sub-scheduler.

3.5.3 Overhead Analysis

In this section, we analyze the overhead in the scheduler caused by invocation of CyberOrgs primitives in the distributed version of CyberOrgs.

- Isolation

  In a cyberorg C_A, if n actors are isolated into a new client cyberorg $C_{A'}$, the resource allocation for these actors has to be recalculated, and this results in overhead in the scheduler. If c_r is the time it takes to recalculate the resource amount for one actor, and c_v is the time it takes to change the value of a time slice for one thread, then the overhead caused by isolation is

  $$n \times c_r + n \times c_v = n \times (c_r + c_v)$$

- Assimilation

  Similar with isolation, assimilation in distributed CyberOrgs system incurs overhead because of resource reallocation for the actors which are in the assimilating cyberorg.

  For a distributed cyberorg C_A which has n actors, the overhead in the scheduler caused by assimilation of C_A is:

  $$n \times c_r + n \times c_v = n \times (c_r + c_v)$$

  where c_r and c_v are respectively the times it takes to recalculate resource allocation for an individual actor and change a value of a time slice in the schedule.

- Local Migration

  Local migration means no actor is migrated to a remote node, and we only need to change the resource allocation for actors of the migrating cyberorg.

  For a distributed cyberorg C_A which has n actors, the overhead in the scheduler caused by migrating C_A to a local cyberorg is:

  $$n \times c_r + n \times c_v = n \times (c_r + c_v)$$

where c_r and c_v are respectively the time it takes to recalculate resource allocation for one individual actor and change a value of a time slice in the schedule.

- Remote Migration

  Remote migration means the destination cyberorg (new host) of the migrating cyberorg is at a different physical node, and the actors in the migrating cyberorg have to be moved to another CyberOrg platform, and be inserted into the schedule there.

  The steps to be carried out in a remote migration are as follows:

  1. Remove n actors from the local scheduler;
  2. Send cyberorg information and actors to the destination cyberorg;
  3. Recalculate the amount of resource that each actor should receive according to the new contract;
  4. Insert n actors to the scheduler queue on the destination node.

  The overhead caused by remote migration of a cyberorg is:

$$n \times c_d + n \times c_r + n \times c_i = n \times (c_d + c_r + c_i)$$

  where n is the number of actors in the migrating cyberorg; c_d is the time it takes to delete an actor from the schedule; c_r is the time it takes to recalculate the amount of resource for one actor; and c_i is the time it takes to insert an actor into the schedule.

From the above analysis, we conclude that the overhead in the distributed scheduler caused by a CyberOrg primitive is linear in the number of local actors involved in the primitive. Because we use a flat scheduler on every individual node, the overhead caused by the switches between actors is linear too.

3.6 Scenarios of Using CyberOrgs

Our CyberOrgs implementation provides a dynamic control for CPU resource in a multi-agent system. Because it enables resource management, it gains overhead from the enabling mechanism. However, there are some scenarios in which we can take advantage of using CyberOrgs. Two example scenarios are described as follows.

3.6.1 Multiple Tasks with Time Constrains

If there are multiple tasks in the system, and some of them need results from others in order to proceed, CyberOrgs can be used for making efficient use of resources. For example, consider a large computation and a small computation are executing concurrently on one computer, and at some stage, the small computation needs the results from the large one. Without resource control, the small computation will likely have to wait for the results of the large computation to become available. Using CyberOrgs, it is possible to allocate more resources to the large computation in order to make it complete faster, so its results are available closer to the time when the small computation requires them.

3.6.2 Mobile Computations

In an open system in which mobile computations are executing in a space of peer-owned resources, remote computations may migrate in order to compete for resources. However, resource competition results in unpredictable delay. By explicitly modeling the ownership of resources, CyberOrgs offers a basis for resource trade. Using CyberOrgs, we can encapsulate local resources and computations into a cyberorg, and any computations which are migrating into the cyberorg have to negotiate with the cyberorg and sign a contract. Because resource trade is specified by the contract, availability of resources is predictable.

3.7 Chapter Summary

In this chapter, we described the implementation of the CyberOrgs model. The system design and implementation details were presented. Our implementation is constructed on Actor Architecture, which is an implementation of Actors. We developed a flat scheduling scheme for scheduling a CyberOrgs hierarchy, and this scheme can be used not only on one node, but also for a distributed CyberOrgs system. The overhead of this flat scheduler was analyzed for both a single node system and a distributed system, and it turns out that the overhead of the scheduler is linear with the number of local actors in the cyberorgs. Although there is certain overhead in our CyberOrgs implementation, it is necessary and profitable to use in some scenarios.

Chapter 4

Interface of CyberOrgs Implementation

4.1 Introduction

Our CyberOrgs implementation provides an application program interface for creating cyberorgs and actors, as well as invoking all the primitives of CyberOrgs. In this Chapter, the API of the CyberOrgs implementation and some example application programs are presented.

4.2 APIs

In this section, the APIs for both creation and CyberOrgs primitives (isolate, assimilate and migrate) are discussed.

4.2.1 APIs for Creation

The implementation supports two types of creation: cyberorg creation and actor creation. Cyberorg creation is called by the GUI or a user program in order to create the first cyberorg in the system; later on, cyberorg creations happen as a result of invocations of the `isolate` primitive. Actor creation is called by an existing actor to create new actors, or by the cyberorg constructor in order to create a facilitator actor.

- Cyberorg creation:

  `CyberOrg createCyberOrg(long eCash, Contract initialContract, String facilitatorClass, Object[] args)`

  where `eCash` is the amount of eCash that is provided to the first cyberorg, `initialContract` is the contract between the first cyberorg and the system, which indicates the amount of resource that the cyberorg receives from the system, and the price of the resource. `facilitatorClass` and `ags` specify the facilitator actor class and the arguments for creating such a facilitator actor.

- Actor creation:

  - `ActorName createActor(ActorName creator, String actorClass, Object[] args)`

    where `creator` is the unique name of the actor which invokes the creation; `actorClass` and `ags` specify the class of the actor being created and the arguments that are used in the actor constructor. This method is called by one actor in order to create a new actor.

  - `ActorName createActor(CyberOrg myCyb, String facilitatorClass, Object[] args)`

    this method is called by the cyberorg constructor to create a facilitator actor. The `myCyb` identifies the cyberorg which invokes this creation; `facilitatorClass` identifies the actor class of the facilitator and `args` specifies the arguments to be used in constructing the facilitator actor.

4.2.2 APIs for CyberOrgs Primitives

CyberOrgs primitives are called by the facilitator actor of the cyberorg.

- Isolation:

  `CyberOrg Isolate(long eCash, ActorName[] actors, Contract newContract)`

where `eCash` is the amount of eCash that is given to the new child cyberorg; `actors` is an array of the existing actors that is isolated to the new child cyberorg; `newContract` is the contract imposed on the new child cyberorg by the host cyberorg. `newContract` specifies the amount of resource that the child cyberorg is to receive, as well as the cost of the resource in terms of the eCash payment to be made.

- Assimilation:

  `CyberOrg Assimilate()`

  this primitive causes assimilation of the cyberorg into its host cyberorg. All contents (actors, resource, eCash) in the assimilating cyberorg are released to the host cyberorg.

- Migration:

  `void Migrate(ActorName facActorOfdesCyberorg, Contract newContract)`

  where `facActorOfdesCyberorg` is the name of the facilitator actor in the destination cyberorg; `newContract` is the contract between the migrating cyberorg and the destination cyberorg, reached as a result of negotiation.

4.2.3 Negotiation

Before migration, a cyberorg needs to negotiate with a potential host cyberorg in order to generate a contract. Negotiation is invoked by the facilitator actor of the cyberorg which is migrating.

- `Contract negotiate(ActorName desFacActor)`

  `desFacActor` is the facilitator actor of the destination cyberorg which the current cyberorg is migrating to.

We assume that there is a discovery service which finds a prospective host. Because resources discovery is not what we focus on, our implementation does not provide an API for "search".

4.3 Examples

Some examples are presented in this section to illustrate how to develop application programs using the API provided by our CyberOrgs implementation. Users can implement applications by extending the `CyberOrg` class and the `FacilitatorActor` class.

4.3.1 System Triggers

This example shows the CyberOrgs primitives can be triggered automatically by some conditions. By subclassing the `FacilitatorActor` class, users can define their own facilitator actor which controls the CyberOrgs primitives according to some specific requirements.

```
public void triggerPrimitives(){
    if (actorList.size()>30)
       triggerIsolation();
    if (actorList.size()<3)
       triggerAssimilation();
    if (myTicks<minRequiredTicks)
       triggerMigration();
}
```

Figure 4.1: Facilitator Actor's Method for Triggering Primitives: If the number of actors in the cyberorg is above a threshold (30), `isolate` is invoked; if the number of actors in the cyberorg is below a threshold (3), `assimilate` is invoked; if the available resource is less than the requirement, `migrate` is invoked

Figure 4.1 shows the method of `triggerPrimitives` which is defined in a facilitator actor of a cyberorg. It checks three conditions and triggers different CyberOrgs primitives accordingly. When there are more than 30 actors in the cyberorg, isolation is invoked; if the number of actors is less than 3, the cyberorg assimilates into its host; when the cyberorg cannot get enough resource for its needs, it finds a potential destination host cyberorg and migrates there to get more resource.

4.3.2 Distributed Weather Forecasting System

Here, we illustrate the use of our API with the help of an example. Consider a distributed weather forecasting system which analyzes weather data, generates alerts when a threatening weather system is entering a region, and takes appropriate actions to address the threat. The computational resources available to such a system may be focused on particular weather systems or population centers. As a weather system moves from one region to another, resources available for analyzing the weather system may need to become available to the regions which require resources for assessing its impact on them.

Two classes of cyberorgs could be used for implementing the distributed weather forecasting system: regional cyberorgs and weather system cyberorgs, both of which are subclassed from the `CyberOrg` class.

- A `regional cyberorg` represents a physical area, and holds actors which analyze and calculate the weather data, in order to develop weather forecasting information for the region. Furthermore, a regional cyberorg manages resources for supporting the weather data computations in it. In the event of localized weather activity, a regional cyberorg would isolate parts of its computation and a sufficient amount of resource dedicated to the affected sub-region to form a new cyberorg with independent control. After the weather threat moves away from the sub-region, or all computations for the sub-region have completed, the cyberorg for the sub-region may assimilate into the cyberorg for the larger region, relinquishing independent control of the resource.

- A `weatherSys cyberorg` represents a weather system which moves between regions, and it carries an amount of eCash dedicated to understanding the weather system. A weather system cyberorg may migrate from one regional cyberorg to another, taking the eCash with it to support the region which is facing the weather system. Specifically, on arriving in a regional cyberorg, the weather system cyberorg would isolate part of its eCash into a new cyberorg,

which would migrate out to the regional cyberorg for assimilation in order to release the eCash.

```
public void actionOnArrival(){
 if (theDisaster.effects(reg)) {
  long toOffer=Needs(theDisaster, reg);
  CyberOrg FundsCyberOrg=
   isolate(toOffer,new ActorName[0],defChildContract);
  ActorName facFund=
   FundsCyberOrg.getFacActorName();
  ActorName facReg=
   reg.getFacActorName();
  send(facFund,"triggerFundMigration",facReg);
 }
}

public void triggerFundMigration
    (ActorName destination){
  Contract offerSupport=negotiateWith(destination);
  if (offerSupport != null){
   migrate(destination, offerSupport);
  }
}
```

Figure 4.2: Methods in the Facilitator Actor of a `WeatherSys` Cyberorg: If the weather system affects the region, funds are isolated to a new cyberorg (`FundsCyberOrg`), and the new cyberorg is migrated to the affected regional cyberorg (`reg`)

Figure 4.2 shows two key methods in the facilitator actor of a `WeatherSys` cyberorg. Method `actionOnArrival` is invoked on the weather system cyberorg's arrival into a regional cyberorg. It checks if the region is under threat. If it is, the region's resource needs are assessed, and a sufficient amount of resource is isolated into a new cyberorg (`FundsCyberOrg`), which is then asked to negotiate terms to migrate to the regional cyberorg. Once a contract is negotiated, `FundsCyberOrg` migrates to the regional cyberorg, and assimilates there to make the resource available to the region.

```
public void checkStatus(){
 if (needToDelegate){
  //identify actors to isolate
  //compute eCash to set aside
  Contract defContract=
    new Contract(ticksRate,eCash,ticks,getHost());
  isolate(eCash,regActors,defContract);
 }
 if (!localControlRequired){
  assimilate();
 }
 if (myContract.res<resNeed){
  ActorName destination=
    lookupYellowPageFor(resNeed);
  Contract newContract=negotiateWith(destination);
  if (newContract!=null) {
    migrate(destination,newContract);
  }
 }
}
```

Figure 4.3: Method `checkStatus` in the Facilitator Actor of a `Regional` Cyberorg: If it is necessary to delegate, a new cyberorg is created for the sub-region with sufficient funds; when the delegation is no longer needed, the new cyberorg assimilates; if the available resource is not enough for the region, it can migrate to a host which can satisfy its resource requirement

Figure 4.3 shows `checkStatus` method in the facilitator actor of a `regional` cyberorg. The `checkStatus` method checks whether the threat being faced cannot be handled. If the regional cyberorg cannot handle the threat, sufficient resource has to be set aside for delegated control for a sub-region which may represent a population center. In this case, the original regional cyberorg isolates computational actors along with the necessary amount of eCash to a new regional cyberorg, which represents the sub-region. Once local control is no longer required (for example, a weather system has passed the sub-region), the sub-regional cyberorg assimilates into the cyberorg of the enclosing region. Furthermore, if the resource available through the contract with the current host cyberorg is not sufficient, a regional

cyberorg may negotiate for a better contract with another cyberorg. In other words, although the cyberorgs for sub-regions are created by the cyberorgs for larger regions, the sub-regional cyberorgs are free to migrate somewhere else in order to receive more resource.

4.3.3 Adaptive Quadrature

Adaptive Quadrature is a classic problem in mathematics. In this section we use adaptive quadrature as an example to illustrate how to use CyberOrgs to perform mathematics computing with dynamic resource control.

4.3.3.1 Introduction to Adaptive Quadrature

As shown in Figure 4.4, $f(x)$ is a real-valued function of a real variable, and we seek to compute the value of the integral on a finite interval $a \leq x \leq b$:

$$\int_a^b f(x)dx$$

Physically, this value is the area which lies underneath the curve $f(x)$ in the given interval $[a, b]$.

Adaptive quadrature is an elementary technique to estimate integral values, and it is based on the fundamental additive property of definite integral:

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$$

where c is any point between a and b. The idea of adaptive quadrature is: approximate the two integrals on the right of the above equation to within a predefined tolerance, then the sum of these two integrals gives the desired result; if not, we recursively apply the additive property to $[a, c]$ and $[c, b]$.

In this example, we use the *trapezoid rule* to calculate an integral: the integral value of $f(x)$ on the interval $[a, b]$ can be estimated by the area of the trapezoid

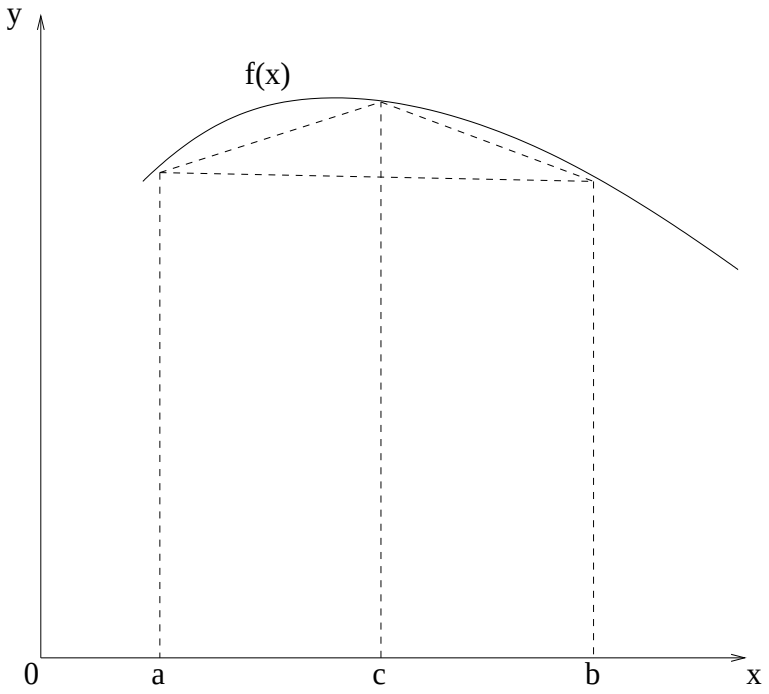

Figure 4.4: Adaptive Quadrature Problem

$(a, b, f(b), f(a))$, which is:

$$(b-a) \times \frac{(f(a)+f(b))}{2}$$

4.3.3.2 Application Actors for Adaptive Quadrature

The first step of implementing an application is designing application actors which carry out the given computation.

In this example, actors are used to calculate adaptive quadrature of a given function $f(x)$. In particular, in order to calculate the value of adaptive quadrature of $f(x)$ in the interval $[a, b]$, we divide this interval at its midpoint m into two subintervals: $[a, m]$ and $[m, b]$. We create two actors to calculate the values of integral on the two subintervals respectively. For instance, as for the subinterval $[a, m]$, we calculate the sum of integrals of its subintervals $[a, n]$ and $[n, m]$ (where n is the midpoint of the interval $[a, m]$) and the value of integral of $[a, m]$, then we get the error e_1 between these two values. Similarly, we calculate the error e_2 for the interval $[m, b]$. If the error is within a specified tolerance ε, the trapezoid area for the

corresponding subinterval is to be reported as an acceptable value of the integral. If the error is greater than the tolerance, a new actor is created, and calculation for the corresponding subinterval is delegated to this newly created actor, which will divide this interval to smaller intervals and calculate the approximate integral values on them.

The `quadrature` method and the `result` method of the application actors are shown in Figure 4.5.

4.3.3.3 Resource Allocation According to Computing Workload

In order to take advantage of the resource management feature of CyberOrgs, we need to put actors into cyberorgs which can allocate resources among them according to their computing workload.

We assume that the actor which is delegated to an interval with higher error needs more resource. We make this assumption because it is necessary to have a criterion for resource management. Although this assumption may not always true, actors which complete their tasks are destroyed right away and all the resource assigned to them are returned to the system, therefore we do not need to worry about assigning more resources to an actor than what are needed.

There are two different ways to allocate resource using CyberOrgs. First, the resource can be distributed by the resource allocation policy inside one cyberorg. Second, we can use multiple cyberorgs to control the resources by delegating resource control to client cyberorgs when necessary.

- Resource allocation policy inside one cyberorg

  In the CyberOrgs model, one cyberorg can have its own resource allocation policy and distribute available resources to the actors inside it. In this example, we put the application actors into one cyberorg which assigns resource to them according to their workload.

  Initially, we create a root cyberorg, and it receives all available resource in the system. Within the root cyberorg, the first application actor is created

```
public void quadrature(double a, double b, ActorName client){
 //get the midpoint of [a,b]:  m
 //calculate error for [a,m]:  erroram
 //calculate error for [m,b]:  errormb
 if ( erroram > ε ) { //create a new actor:  newChildam
  send(newChildam,"quadrature",a,m,newChildam);
 } else {
   result(trapezoid(a,m));
  //report approximation of the integral on [a,m]
 }
 if ( errormb > ε) {
  //create a new actor:  newChildmb
  send(newChildmb,"quadrature",m,b,newChildmb);
 } else {
   result(trapezoid(m,b));
  //report approximation of the integral on [m,b]
 }
}


public void result(double reportValue){
  (numResponses == 0){
   numResponses++;
   partialResponse = reportValue;
 } else {
   myResults = reportValue + partialResponse;
  //report "myResult" to the actor which created this actor
  destroy("task completed!"); //destroy current actor
 }
```

Figure 4.5: Application Actors' `quadrature` and `result` Methods in the Adaptive Quadrature Example

for computing the adaptive quadrature. According to the algorithm which is described in the previous section, new actors are created inside the root cyberorg to pursue a final result which is accurate enough (the error is less then the tolerance ε). At the time of creation of each new actor, certain amount of resource is dedicated to it by the cyberorg.

A cyberorg has its own local resource allocation policy, and this policy is car-

ried out by the facilitator actor in this cyberorg. In this example, `Facilitator` class is extended in order to asses the workload for each actor and assign resource to it.

Specifically, the facilitator keeps track of the average error of all the actors in the system, as well as the average amount of resources assigned to the actors. For the first application actor, the facilitator assigns it a predefined percentage of resource (e.g. 10%). When a new actor is created, the facilitator compares the new actor's error with the average error, calculates the ratio and distributes resource to the new actor according to the ratio.

Figure 4.6 shows the `resourceAllocate` method in the facilitator class, which takes as parameters the name of the new actor, and the error of the integral on the new actor's interval. An amount of resource can be assigned to the newly created actor by invoking this method.

If there are multiple instances of adaptive quadrature computation in the system, we can use this approach to allocate resources to them by encapsulating each instance to one cyberorg.

```
public void resourceAllocate(ActorName newActor, double error){
  ratio = error/averageError;
    //compute the ratio of error to
    //the average error value of all actors
  myTicks = averageTicks * ratio;
  myTicksRate = averageTicksRate * ratio;
    //calculate the amount of resources
    that is assigned to the new actor
  hostCyberOrg.resourceAllocate(newActor,myTicks,myTicksRate);
    //tell host cyberorg to carry out resource allocation
  /* update the value of averageError */
  /* update the value of averageTicks and averageTicksRate */
}
```

Figure 4.6: Method `resourceAllocate` in the Facilitator Actor of the Cyberorg in the Adaptive Quadrature Example (Resource Allocation Policy)

```
public void resourceAllocate(ActorName newActor, double error){

  ratio = error/averageError;
    //compute the ratio of error to the average error
    //value of all actors

  if (ratio > threshold){
   myeCash = averageeCash * ratio;
   /* generate a contract for isolation */
   hostCyberOrg.isolate(eCash,newActor,myContract);
     //tell the host cyberorg to isolate the new
     //actor to a child cyberorg

   /* update the value of averageError */
   /* update the value of averageeCash */
  }

 else{

   myTicks = averageTicks * ratio;
   myTicksRate = averageTicksRate * ratio;
    //calculate the amount of resources that is
    //assigned to the new actor
 hostCyberOrg.resourceAllocate(newActor,myTicks,myTicksRate);
    //tell the host cyberorg to carry out the resource allocation

  /* update the value of averageError */
  /* update the value of averageResource */
   }
}
```

Figure 4.7: Method `resourceAllocate` in Facilitator Actor of the cyberorg in the Adaptive Quadrature Example (Multiple Cyberorgs)

- Multiple cyberorgs

  Instead of developing a particular resource allocation policy, another method for controlling resources is using multiple cyberorgs. Each of the cyberorgs uses the resource allocation policy which is described in previous approach. When a new application actor is created inside one cyberorg, this cyberorg

may isolate it to a new client cyberorg if the new actor's error is more than a threshold.

Similar with the previous method, we still extend the `Facilitator` class to control resource, but we use the `isolate` primitive to encapsulate the new actor (which has more workload) to a new cyberorg in order to guarantee resource supply. In particular, if the ratio of the new actor's error to the average error exceeds some threshold, we set aside some eCash and isolate this new actor to a new cyberorg; and if not, we use the resource allocation policy of the original cyberorg to assign resources to the new actor.

Figure 4.7 shows the `resourceAllocate` method in the `Facilitator` class.

4.4 Chapter Summary

In this chapter, we described the APIs of our CyberOrgs implementation, including cyberorg creation, actor creation, as well as CyberOrgs primitives. We also show how to use these APIs to control resource using multiple examples.

Chapter 5

Experimental Results

5.1 Introduction

Experimental results are presented in this Chapter. Experiments using simulated workload are carried out to analyze the overhead of our CyberOrgs implementation, the results about performance analysis are presented in Section 5.2. In Section 5.3, the actual application programs using CyberOrgs are described, as well as the results analysis. Experiments on CyberOrgs migration are carried out to measure the network delay of migrating a cyberorg, and the results are presented in Section 5.4.

5.2 Performance Analysis

Experiments have been carried out to look at the overhead of using cyberorgs. Two types of the flat scheduler which are based on `suspend/resume` and priority are compared. The experiments and results are presented in this section.

5.2.1 Experiment Design

In order to show the efficiency of the flat scheduler, we compared the performance of our scheduler with the Java Virtual Machine's default thread scheduler by having them schedule the same number of threads which carry out identical computation tasks.

For different numbers of threads (from 10 to 1000), several types of experiments were carried out:

1. With cyberorgs:

   In this experiment, we scheduled all (actor) threads in cyberorgs organized as a tree. The experiment began with a root cyberorg with an initial number of threads. Thereafter, cyberorg isolation or creation of new threads was invoked randomly until the total number of threads reached the predefined number. Random time slices were assigned to each thread in the system. We performed 5 runs and calculated the average execution time for each experiment, and the standard deviations were also calculated.

2. Fair scheduler (max, min, mean):

   In these three experiments, there was no cyberorg, and each thread was scheduled for a fixed number of time slices in one scheduling cycle. For fair comparison, we carried out three groups of experiments using three fixed time slices respectively, which were the maximum, minimum, and mean value of all time slices we got from the experiments on cyberorgs for corresponding number of threads.

3. No scheduler:

   There was no cyberorg or any custom schedulers in this experiment, and we allowed the JVM default scheduler to schedule the threads in the system.

4. In one thread:

   In this experiment, all workloads of corresponding numbers of threads were carried by one thread. [1]

5.2.2 Context Switching Using `Suspend/Resume`

The principle of the scheduler which uses `suspend/resume` is: suspend every thread in the system immediately after its creation, and resume the threads one by one for

[1] We show the "one thread" here just for reference instead of comparison, because it does not represent concurrent computing, as the other experiments do.

certain amounts of time according to the time slices that the threads receive.

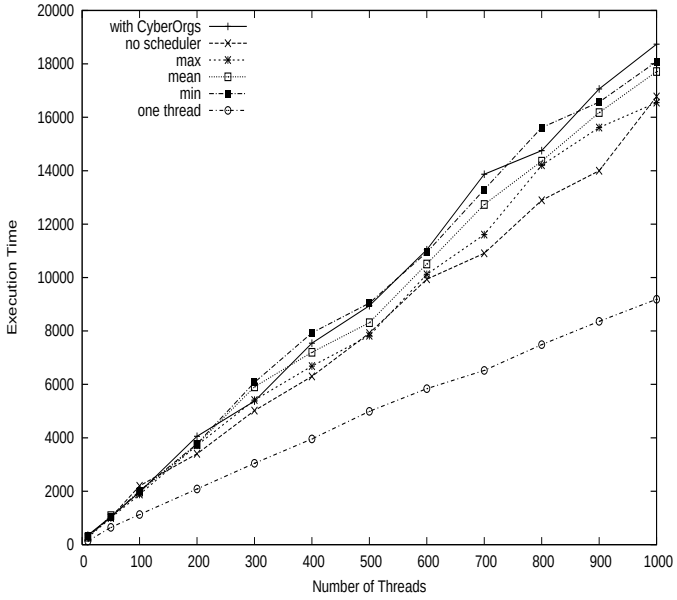

Figure 5.1: Performance Comparison of Scheduling Choices in Flat Scheduler Which Uses `Suspend/Resume`

As shown in Figure 5.1, the execution times of all types of experiments appear to be almost linear with the number of threads in the system. The experiment with no scheduler has the best performance, because the threads are not suspended or resumed; in Java, `suspend` and `resume` are time consuming. For the three fair scheduling experiments, the one with maximum time slice has the best performance; on the contrary, the one with minimum time slice has the worst. The smaller the time slice, the more times we need to switch from one thread to another in the whole process, and because the context switches are based on suspend and resume, the overhead increases when the number of switches increases. Compared with the fair schedulers, the scheduler with cyberorgs does not have significant overhead. Sometimes it is better, but sometimes it is worse than the fair schedulers. The reason is that the scheduler with cyberorgs constructs and maintains a random cyberorgs tree, and the time slice for each thread is random too. Although there is some uncertainty in the scheduler with cyberorgs, it does not cause prohibitive overhead. Table 5.1 shows the data from the experiments on the scheduler based on `suspend/resume`.

	With cyberorgs				Fair scheduler						No	One
#	Height	Cyberorgs	Time	Stdev	Max	Time	Mean	Time	Min	Time	scheduler	thread
10	2	4	356	37.8	14	272	8	280	2	334	319	137
50	4	17	1040	94.5	183	999	33	1087	2	1022	1020	646
100	3	15	1967	100.4	146	1878	17	1969	2	2016	2201	1129
200	4	27	4058	85.7	250	3720	15	3750	2	3775	3399	2083
300	5	40	5372	96.9	370	5412	19	5908	2	6074	5017	3047
400	5	67	7544	108.5	356	6685	21	7202	2	7931	6299	3960
500	5	59	8946	94.5	239	7823	13	8313	2	9043	7922	4993
600	5	71	11040	191	352	10121	11	10507	2	10943	9938	5841
700	5	102	13866	85.2	390	11607	11	12736	2	13291	10906	6524
800	5	74	14754	253.7	330	14203	11	14359	2	15614	12892	7460
900	6	129	17061	423.1	634	15617	16	16177	2	16568	13998	8359
1000	6	140	18736	684.9	324	16548	14	17715	2	18087	16781	9188

Table 5.1: Performance Comparison of Scheduling Choices in Scheduler Which Uses `Suspend/Resume`: Time is in Milliseconds; Cyberorgs is the Final Number of Cyberorgs in the System; Height is the Final Height of the Cyberorgs Tree

5.2.3 Context Switching Using Priority

In this section, the scheduler still has a flat structure, but the context switching is based on priority instead of `suspend/resume`. In Java, changing the priority of a thread is more efficient than suspending or resuming a thread. We assign an initial priority to each thread when it is created. In one scheduling cycle, we increase each scheduled thread's priority for its time slice.

Figure 5.2 shows the results of several types of experiments on the flat scheduler using priorities for scheduling, and the results are similar to what we get in the scheduler based on `suspend/resume` (as shown in Section 5.2.2).

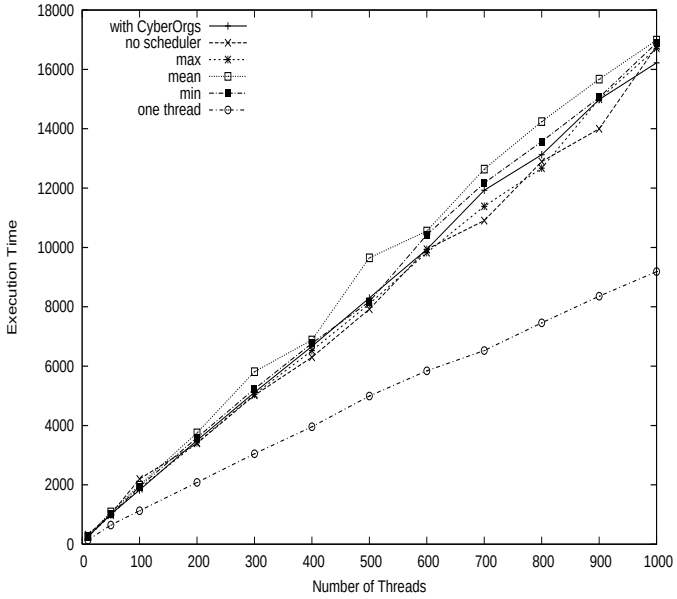

Figure 5.2: Performance Comparison of Scheduling Choices in Flat Scheduler Which Uses Priority

The disadvantage of the priority scheduler is that the priority is not precisely specified in Java [12], and the relationship between priority and scheduling is unclear. For example, when a thread's priority is decreased, it may not be suspended by the system right away. Furthermore, when the time slice is very large, there is more possibility that a thread complete in the middle of the time slice, and if so, we lose the control for the rest of time in this time slice. Therefore the flat scheduler based on priority may not control resources precisely.

	With cyberorgs				Fair scheduler						No	One
#	Height	Cyberorgs	Time	Stdev	Max	Time	Mean	Time	Min	Time	scheduler	thread
10	1	1	251	35.1	45	252	25	254	3	267	319	137
50	4	11	1011	28.3	304	981	63	1005	2	1086	1020	646
100	4	17	1828	59.6	240	1878	28	1941	2	1990	2201	1129
200	5	28	3499	166.8	640	3418	45	3589	2	3752	3399	2083
300	5	29	5113	89.8	528	5051	37	5231	2	5812	5017	3047
400	5	59	6666	151	552	6525	32	6769	2	6882	6299	3960
500	5	70	8297	136.3	504	8100	30	8173	2	9650	7922	4993
600	4	42	9934	59.4	640	9823	15	10429	2	10554	9938	5841
700	5	55	11927	302.6	560	11387	13	12168	2	12640	10906	6524
800	6	82	13126	393	1280	12670	25	13570	2	14240	12892	7460
900	5	100	14985	212.7	800	14996	19	15052	2	15670	13998	8359
1000	6	131	16222	185.5	960	16712	19	16892	2	16985	16781	9188

Table 5.2: Performance Comparison of Scheduling Choices in Scheduler Which Uses Priority: Time is in Milliseconds; Cyberorgs is the Final Number of Cyberorgs in the System; Height is the Final Height of the Cyberorgs Tree

Table 5.2 shows the results of the experiments on the scheduler based on priority.

From the results of the experiments, we conclude that no significant overhead occurs in using the flat scheduler to schedule a tree of cyberorgs. Specifically, the overhead is unrelated to the number of cyberorgs and the height of the cyberorgs tree. We reach the same conclusion from both the `suspend/resume` approach and the priority approach, but we choose the `suspend/resume` approach in our implementation because it is more accurate than the priority one.

5.3 Application Experimental Results

In this section, we present and analyze the results of the *adaptive quadrature* example which was described in Section 4.3.3.

We take the function $f(x) = x\sin(\frac{1}{x}), x \in [0, 1]$ as an example, i.e., we calculate $\int_0^1 x\sin(\frac{1}{x})dx$ (we define f(0)=0). Using the CyberOrgs system, we can allocate the processor time resource to each actor according to its workload. We compare the performance of two approaches (Section 4.3.3) using CyberOrgs system and the fair scheduling. In the experiment "One Cyberorg", we use one cyberorg to allocate resource to the application actors for the adaptive quadrature calculation. In the "Multiple Cyberorgs" experiment, we use cyberorgs hierarchy to control resource, i.e., if an actor's error exceeds a threshold, `isolate` is triggered to encapsulate this actor to a new cyberorg and delegate resource control. In the fair scheduling experiments, we schedule all application actors in a fair scheduler, which assigns the same time slice (the maximum, minimum, and mean values of the time slices in "Multiple Cyberorgs") for every application actor.

The experimental results are shown in Table 5.3 and Figure 5.3. We use the minus logarithm of the error tolerance for the x axis in Figure 5.3, in order to disperse the sample points. Points with higher x values represent adaptive quadrature computation with smaller error tolerance, which requires more calculation. The results illustrate that compared with the fair scheduling, there is no significant overhead caused by the hierarchical resource control in CyberOrgs system, which is consistent

Error Tolerance	One Cyberorg	Multiple Cyberorgs	Fair Scheduling Max	Mean	Min
0.1	2401	2349	2385	2380	2572
0.075	2630	2378	2403	2587	2603
0.05	2420	2389	2417	2377	2610
0.025	2785	2798	2577	2589	2679
0.01	3008	3047	2678	2735	2905
0.0075	3549	3695	2751	3267	3729
0.005	4772	4905	3490	4351	4790
0.0025	5748	6637	5401	6210	6968
0.001	7026	7930	5829	6969	8120
0.00075	7315	8947	6075	7439	8997
0.0005	7408	9723	6970	7521	9560
0.00025	8075	10011	7324	9991	10373
0.0001	9820	11640	8035	10987	11377

Table 5.3: Experimental Results for the Adaptive Quadrature Example

with our performance analysis results. The performance of "Multiple Cyberorgs" is worse than the "One Cyberorg", because the former enforces a hierarchical resource control, while the latter does not have a hierarchy.

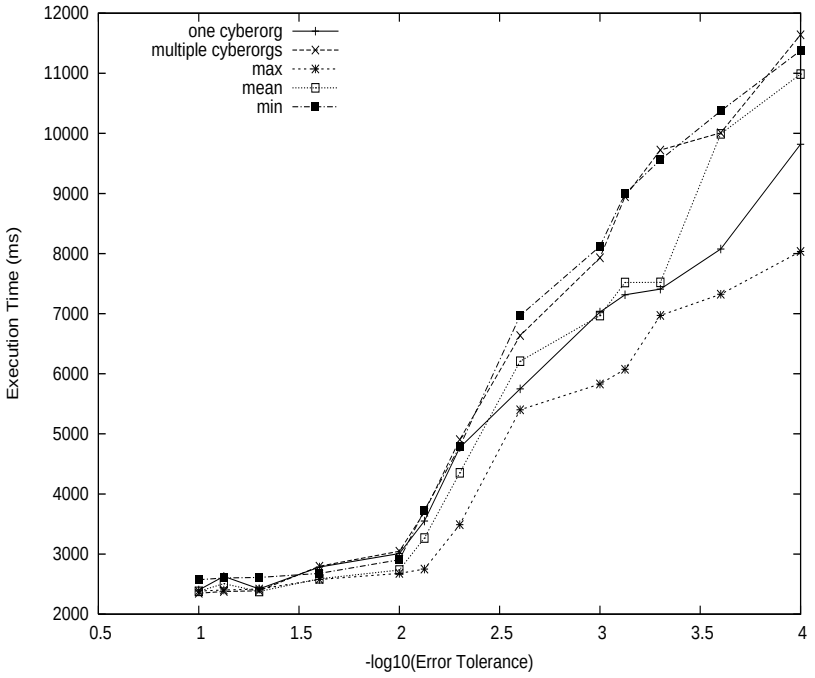

Figure 5.3: Experimental Results for the Adaptive Quadrature Example

5.4 Network Delay Experimental Results

Experiments have been carried out on the distributed CyberOrgs system to test the network delay of the migration.

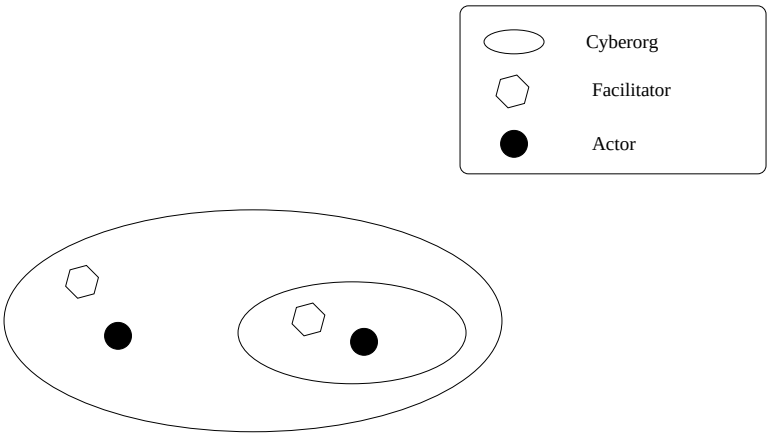

Figure 5.4: The Structure of the Migrating Cyberorg

For example, we migrate a cyberorg (as shown in Figure 5.4) from one node to another node. The network delay of this remote migration is shown in Table 5.4.

The results illustrate that it takes 610 milliseconds to send the structure information of the example cyberorg to a remote node, and to extract the information there. Migrating a single actor and inserting it into the remote schedule takes 547 milliseconds. Compared with migrating an actor, migrating cyberorgs is not very costly.

Total Time for the Migration (ms)	Delay for Migrating an Actor (ms)	Delay for Migrating the Cyberorgs (ms)
2798	547	610

Table 5.4: Network Delay of Migrating the Example Cyberorg

Both message passing and cyberorg migration consume network bandwidth resource, and this fact motivates our future work of network bandwidth resource control using CyberOrgs, which makes the network delay manageable.

5.5 Chapter Summary

In this Chapter, we analyze the performance of using CyberOrgs system through both simulated workload and actual application workload. Network delay of migrating a cyberorg is also analyzed. The conclusions are as follows:

- CyberOrgs implementation gains overhead from the resource acquisition and control.
- The overhead can be reduced by increasing the granularity of resource control, and this is the trade-off between the overhead and resource control.
- Hierarchical resource control in CyberOrgs system does not cause significant overhead.
- In comparison with migrating an actor, migrating a cyberorg is not very costly.

Chapter 6

Future Work

6.1 Introduction

The first version of CyberOrgs implementation has been developed, and it can be used for resource coordination in a multi-agent system. At present our CyberOrgs implementation aims at processor time control, including single processor time and distributed processor time.

In this Chapter, some future directions are presented.

6.2 Internally Distributed CyberOrgs

An internally distributed cyberorg is located on multiple nodes. In other words, one part of such an cyberorg can exist on one node and other parts of the same cyberorg may exist on other nodes. If there is at least one internally distributed cyberorg existing in the CyberOrgs system, we call the system an *internally distributed CyberOrgs System.*

6.2.1 Cyberorg

A cyberorg can be physically located on one or multiple nodes. Whether a cyberorg is internally distributed or not depends on the type of resources it holds. For example, cyberorg C_A is created on node A, and at the time of its creation, it is assigned certain amount of processor time on node A, as well as some resources on

another node, B. In this case, cyberorg C_A has to be internally distributed in order to manage the distributed resources it holds.

There are two different ways to implement an internally distributed cyberorg, one facilitator per cyberorg or multiple facilitators. The former uses one facilitator to manage all resources the cyberorg possesses, which are physically located on several nodes, and all cyberorg primitives are handled by this single facilitator. The latter uses multiple facilitators for one cyberorg, and the number of facilitators depends on the number of nodes on which this cyberorg is located. One facilitator is only in charge of the resources located on one physical node, and the cyberorg primitives are handled by multiple facilitators.

Figure 6.1 and Figure 6.2 show the two implementation approaches using an example: one cyberorg which is distributed on two nodes: A and B.

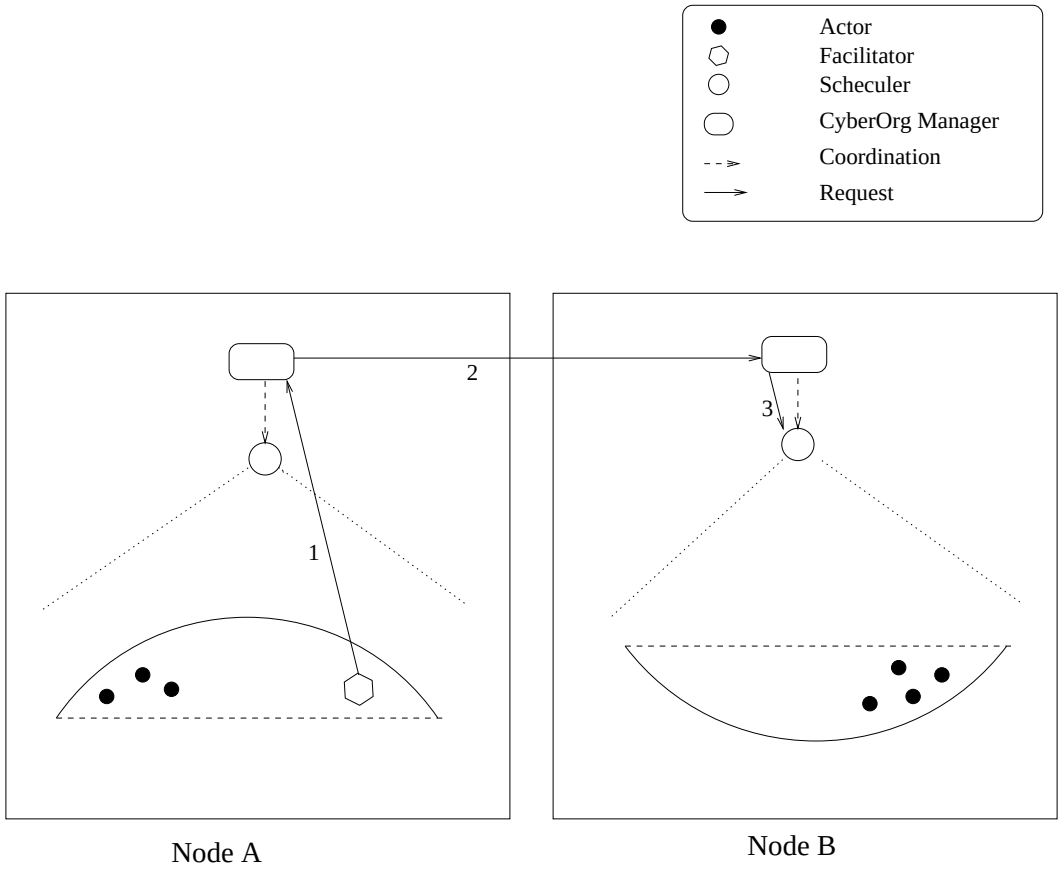

Figure 6.1: Internally Distributed Cyberorg Implementation : Approach 1 (One Facilitator per Cyberorg)

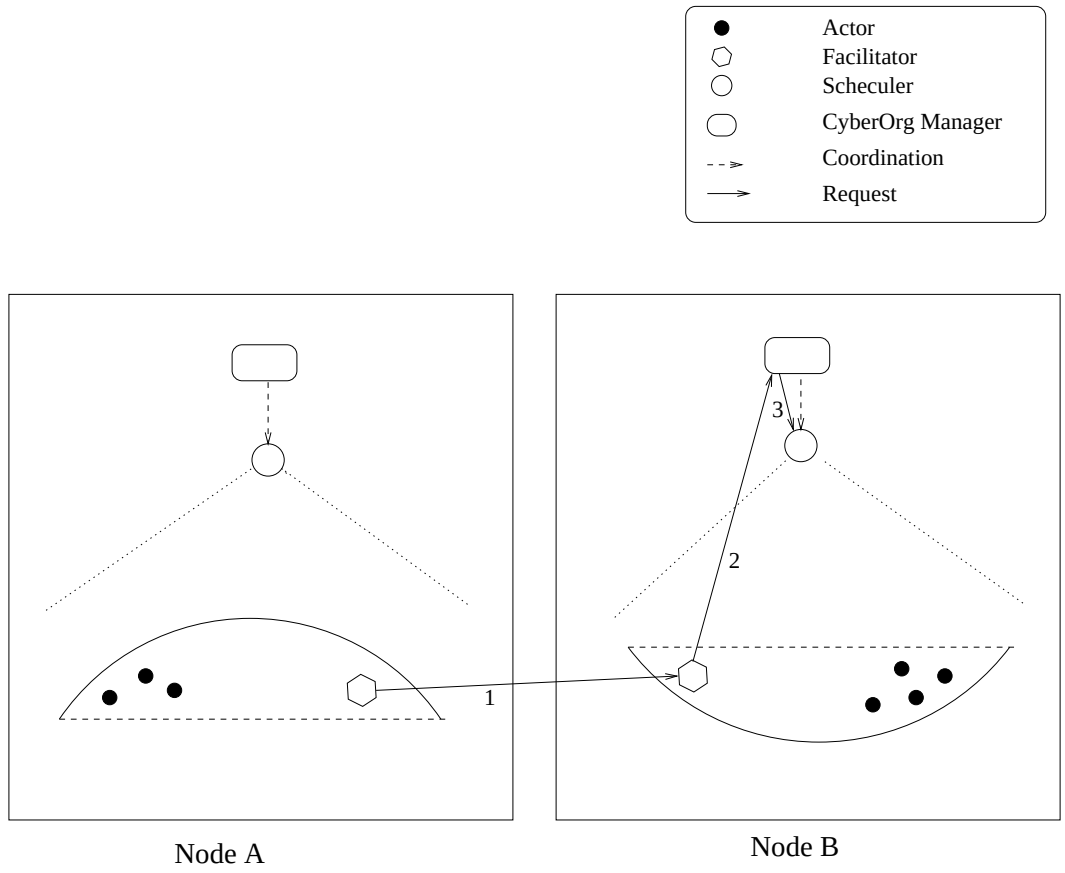

Figure 6.2: Internally Distributed Cyberorg Implementation : Approach 2 (Multiple Facilitators)

These two approaches are different in implementing the CyberOrg primitives. We take the isolation of an internally distributed cyberorg as an example. Suppose in an cyberorg C_1 which is distributed on m nodes, n actors are isolated to create a new cyberorg C_2, and these n actors are located at m' nodes ($m' < m$). The operations for the isolation in the two implementation approaches are as follows:

- Approach 1: One facilitator per cyberorg

    1. The facilitator calculates resource amounts for all n actors.
    2. The facilitator informs the local CyberOrg Manager about the resource amounts.
    3. The local CyberOrg Manager sends the calculated resource amounts to CyberOrg Managers on the $m' - 1$ nodes.

	One facilitator per cyberorg	Multiple facilitators
number of facilitators	1	m
computation overhead caused by one primitive	$n \times c_r$ (on one node)	$n \times c_r$ (on m' nodes)
communication overhead caused by one primitive	$(m' - 1) \times c_c$ (between facilitator and CyberOrg Manager)	$(m' - 1) \times c_c$ (between facilitators)
modification overhead caused by one primitive	$n \times c_v$	$n \times c_v$

Table 6.1: Comparison of the Overhead in Two Implementation Approaches for Internally Distributed CyberOrgs

4. The CyberOrg Managers on m' nodes change the schedules on their respective nodes according to the calculated resource amounts.

- Approach 2: Multiple facilitators

1. The facilitator which receives the primitive request sends messages to $m' - 1$ facilitators to invoke corresponding primitives on them.
2. On each of the m' nodes, calculations about the resource availability are carried out only for local actors on that node.
3. Each facilitator sends calculated resource amounts to the local CyberOrg Manager.
4. Each CyberOrg Manager changes corresponding schedule on its node.

Table 6.2.1 shows the comparison of the overhead caused by invoking a CyberOrg primitive in cyberorg C_1 using these two implementation approaches. Here n is the number of actors that are involved in the primitive, and m' is the number of nodes at which these n actors are located ($m' \leq n$). c_r, c_c, and c_v are respectively the overheads caused by recalculating resource allocation, sending a message to a remote node, and changing a value of one time slice in the schedule.

It is obvious that second approach (with multiple facilitators) performs better than the first approach (one facilitator), because in the Approach 1, the number of messages sent to CyberOrg Managers (which are most likely busy) is fewer than in

Approach 2, and the resource allocation computation in Approach 2 is distributed to each of the nodes involved in the primitive. Moreover, using a central facilitator to control distributed resources is hard to implement. We adopt Approach 2 in the analysis of internally distributed CyberOrgs.

6.2.2 CyberOrg Manager

The CyberOrg Managers in an internally distributed CyberOrgs system are similar to CyberOrg Managers in the case of distributed CyberOrgs (Section 3.5.2). There is one CyberOrg Manager on each node, which is responsible for delivering resources to the actors on this node.

Because it is the CyberOrg Manager that keeps track of the structure of cyberorgs, for each internally distributed cyberorg, the CyberOrg Manager maintains a name table of facilitator actors which belong to other parts of this cyberorg that exist on different nodes.

6.2.3 Scheduler Manager

The Scheduler Manager in internally distributed CyberOrgs is identical to the one for distributed CyberOrgs (Section 3.5.1).

6.2.4 Overhead Analysis

In this section, we analyze the overhead in the scheduler caused by invocation of CyberOrgs primitives in the internally distributed CyberOrgs.

- Isolation

  In an internally distributed cyberorg C_A, the overhead in the scheduler is not only caused by resource reallocations, but also by communications between sub-schedulers. If n actors in the cyberorg C_A are isolated into a new client cyberorg $C_{A'}$, and these n actors are located on m' nodes, then the overhead

on the schedulers resulting from the isolation primitive is:

$$n \times c_r + n \times c_v + (m' - 1) \times c_c$$

where c_r is the time it takes to recalculate the resource for one actor; c_v is the time it takes to change a value of the time slice for one thread in the schedule; and c_c is the cost of communication. Because the actors involved in the isolation are located on m' nodes, the facilitator which receives the isolation request has to send messages to the facilitators in other parts of the cyberorg, in order to invoke resource reallocation on those nodes.

- Assimilation

  Similar with isolation, assimilation in distributed CyberOrgs system incurs overhead because of resource reallocation for the actors which are in the assimilating cyberorg.

  In an internally distributed cyberorg C_A with n actors, the overhead for assimilation is:

  $$n \times c_r + n \times c_v + (m - 1) \times c_c$$

  where c_{1r} and c_v are respectively the times it takes to recalculate resource allocation for one individual actor and change a value of a time slice in the schedule, and c_c is the time it takes to send a message from one facilitator to another facilitator.

- Local Migration

  Local migration means no actor is migrated to a remote node, and we only need to change the resource allocation for actors of the migrating cyberorg.

  For an internally distributed cyberorg C_A with n actors which are distributed on m nodes, the overhead from migrating C_A to another internally distributed

cyberorg C_B[1]:

$$n \times c_r + n \times c_v + (m-1) \times c_c$$

where c_r and c_v are respectively the time it takes to recalculate resource allocation for one individual actor and change a value of a time slice in the schedule, and c_c is the time it takes to send a message from one facilitator to another facilitator.

- Remote Migration

  Remote migration means the destination cyberorg (new host) of the migrating cyberorg is at a different physical node, and the actors in the migrating cyberorg have to be moved to another CyberOrg platform, and be inserted into the schedule there.

  The steps to be carried out in a remote migration are as follows:

  1. Remove n actors from the local scheduler;
  2. Send cyberorg information and actors to the destination cyberorg;
  3. Recalculate the amount of resource that each actor should receive according to the new contract;
  4. Insert n actors to the scheduler queue on the destination node.

  If the cyberorg which is migrating is internally distributed, there is additional overhead caused by the communication between facilitators (the facilitator which receives the primitive request has to send it to other facilitators in other parts of the cyberorg).

  As a result, the overhead caused by a remote migration of an internally distributed cyberorg is:

$$n \times c_d + n \times c_r + n \times c_i + (m-1) \times c_c$$

[1]Here we mean local migration, so cyberorg C_B must be internally distributed, and it must have m or more than m hosts which include the nodes where C_A is located.

where n is the number of actors in the migrating cyberorg; c_d is the time is takes to delete an actor from the local schedule; c_r is the time it takes to recalculate the resource allocation for one actor; c_i is the time is takes to insert an actor into the schedule; and c_c is the time it takes to send a message to another node where some of the actors are involved in the migration are located.

6.3 Network Resource Control

Network bandwidth is an important type of resource in multi-agent systems, because it is necessary for an agent to communicate with other agents which are located on a different node.

The current version of CyberOrgs implementation can be generalized in order to control the network bandwidth resource. For the processor resource control, we use processor time as a measure, which is obvious. Similarly, for network bandwidth, it is necessary to abstract the resource to a measurable number.

Network bandwidth is a resource that is required when an actor is going to send message to another actor which is located on a different node. Intuitively we can use the number of messages an actor or a cyberorg can send within a fixed period of time (say, per second) to represent the amount of network bandwidth it holds. Initially all of the network resources belong to the root cyberorg, which assigns part of the resource to the actors and the client cyberorgs it hosts according to the local resource allocation policy and the contracts. The network resource is represented by an upper limit of messages being sent. Sending messages consumes network resources. When a cyberorg runs out of network resources, all of actors it hosts are not be able to send messages to a remote actor.

The flat scheduling approach can be generalized to control network resource. Specifically, a message scheduler can be used to manage the outgoing messages in a CyberOrgs system. The message scheduler has a flat structure, and it maintains received messages in a queue and send them one by one. It is the CyberOrg Manager

that maintains the hierarchy of cyberorgs, enforces the hierarchical resource control, and inserts each cyberorg's outgoing messages to the message scheduler's queue. Messages from a cyberorg with more network resource have shorter delay.

6.4 Chapter Summary

In this Chapter, future directions of our research work were described, including internally distributed CyberOrgs and network bandwidth resource control. Two approaches of implementing internally distributed CyberOrgs were compared, and their overhead were analyzed. Network resource control using CyberOrgs was described, and our flat scheduling approach can be generated for controlling network bandwidth.

Chapter 7

Conclusion

In a multi-agent system distributed over a peer-owned network, agents share a computational space where the resources connected through the network are owned by independent peers. Because every agent requires resources to support its computations, resource acquisition and control is an important concern in deploying multi-agent systems.

The fundamental motivation for the research work presented in this thesis is the growing demand for an efficient and scalable mechanism of resource management in multi-agent systems. Hierarchical structure is adopted by many resource management models (CyberOrgs model is one of them), but enforcing a hierarchical resource coordination is very costly. In this thesis, we use CyberOrgs model as an example to illustrate that the hierarchical resource control can be implemented in an efficient manner.

Although the CyberOrgs model organizes resources and agents in a hierarchy, the amount of resource allocated to each agent can be calculated and enforced in a resource schedule which has a flat structure. By flattening the resource schedule, we reduce the prohibitive overhead caused by the hierarchical control. Experiments have been carried out on our CyberOrgs implementation, and the results illustrate our implementation does not cause significant overhead in comparison with the traditional round-robin scheduling. Moreover, the overhead caused by enforcing the resource control in our CyberOrgs implementation is manageable by adjusting the granularity of control.

The resource we deal with in our implementation is the processor time resource, but this approach can be generalized for other types of resources.

References

[1] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, 1986.

[2] Gul A. Agha, Svend Frϕund, Woo Young Kim, Rajendra Panwar, Anna Patterson, and Daniel Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel & Distributed Technology: Systems & Technology*, 1(2):3–14, 1993.

[3] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[4] Mark Astley. The actor foundry. *Manual of the Actor Foundry, version 0.2.0*, 1999.

[5] William D. Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusetts Institute of Technology, 1981.

[6] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems—Concepts and Design*. Addison-Wesley Publishing Company, second edition, 1994.

[7] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciaran Bryce. A resource management api for java platform. *SMLI TR-2003-124*, 2003.

[8] Grzegorz Czajkowski and Thorston von Eichen. Jres: A resource accounting interface for java. *13th ACM OOPSLA, Vancouver, BC*, 1998.

[9] Frank Dignum and Mark Greaves. Issues in agent communication: An introduction. In *Issues in Agent Communication*, pages 1–16, 2000.

[10] James E.White. Telescript technology: The foundation for the electronic marketplace. *Technical report, General Magic Inc., Mountainview, CA*, 1994.

[11] Bryan Ford and Sai Susarla. Cpu inheritance scheduling. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 91–105. ACM Press, 1996.

[12] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 1996.

[13] Irene Greif. Semantics of communicating parallel processes. Technical report, Massachusetts Institute of Technology, 1975.

[14] Carl E. Hewitt. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*. PhD thesis, Massachusetts Institute of Technology, 1971.

[15] Carl E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3), June,1977.

[16] Carl E. Hewitt and Peter de Jong. Open systems. In John Mylopoulos Michael L. Brodie and Joachim W. Schmidt, editors, *On Conceptual Modelling*, chapter 6, pages 147–164. Springer Verlag, 1984.

[17] Henry Hexmoor, Cristiano CasteIfranchi, and Rino Falcone. *Agent Autonomy*. Kluwer Academic Publishers, Boston, Dordrecht, London, 2003.

[18] Raj K. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley, New York, USA, 1991.

[19] Nadeem Jamali. *CYBERORGS: A Model for Resource Bounded Complex Agents*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[20] Nadeem Jamali, Prasanna V. Thati, and Gul A. Agha. An actor-based architecture for customizing and controlling agent ensembles. *IEEE Intelligent Systems*, 14(2):38–44, March/April 1999.

[21] Nadeem Jamali and Xinghui Zhao. A scalable approach to multi-agent resource acquisition and control. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2005)*, pages 868–875, Utrecht, Netherlands, July 2005. ACM Press.

[22] Myeong-Wuk Jang and Gul A. Agha. Efficient communication in multi-agent systems. *Software Engineering for Scale Multi-Agent Systems III, Lecture Notes in Computer Science 3390, Springer-Verlag*, 2005.

[23] William A. Kornfeld and Carl E. Hewitt. The scientific community metaphor. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 311–320. Kaufmann, San Mateo, CA, 1988.

[24] Open System Laboratory. The actor foundry: A java-based actor programming environment. *Available for download at(http://osl.cs.uiuc.edu/foundry)*, 1999.

[25] Open System Laboratory. The actor architecture. *Available for download at(http://osl.cs.uiuc.edu/aa)*, 2004.

[26] Manoj Lal and Raju Pandey. A scheduling scheme for controlling allocation of cpu resources for mobile programs. *Autonomous Agents and Multi-Agent Systems*, 5, 2002.

[27] John Lehoczky, Lui Sha, and Ye Ding. The rate-monotonic scheduling algorithm: Exact characterization and average behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[28] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[29] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[30] Thomas W. Malone. Modeling coordination in organizations and markets. In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 151–158. Kaufmann, San Mateo, CA, 1988.

[31] Carl Manning. *ACORE: The design of a core actor language and its compiler*. PhD thesis, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1987.

[32] Robin Milner. *Communication and Concurrency*. Prentics Hall, 1989.

[33] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[34] Luc Moreau and Christian Queinnec. Design and semantics of quantum: a language to control resource consumption in distributed computing. *Usenix Conference on Domain-Specific Languages(DSL'97)*, pages 183–197, 1997.

[35] Luc Moreau and Christian Queinnec. Distribued and multi-type resource management. *ECOOP'02 Workshop on Resource Management for Sage Languages,Malaga, Spain.*, 2002.

[36] Reid Simmons. Towards reliable autonomous agents. In *Proc. of the AAAI Spring Symp. on Lessons Learned from Implememted Software Architectures for Physical Agents*, Stanford, CA, 1995.

[37] Niranjan Suri. State capture and resource control for java: The design and implementation of the aroma virtual machine. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[38] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, and Timothy S. Mitrovich. An overview of the nomads mobile agent system. In *Proceedings of ECOOP'2000, Nice, France, 2000*, 2000.

[39] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, 1996.

[40] Jarle G. Hulaas Walter Binder and Alex Villazon. Portable resource control in java: The j-seal2 approach. *16th ACM OOPSLA, Tampa Bay, FL*, 2001.

[41] Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[42] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley and Sons, Chichester, England, 2002.

Appendix A: Code for Adaptive Quadrature

```
/*
 * QuadActor.java
 *
 * Created on June 21, 2005, 11:20 AM
 */

package aa.application.AdaptiveQuad;


import aa.core.Actor; import aa.core.ActorName;
import aa.core.CreateActorException;
import java.lang.Math;
import aa.core.ActorThread;


/**
 *
 * @author  xinghuizhao
 */
public class QuadActor extends Actor{

    int numResponses; //number of response it has received

    double partialResponse; //temporary result

    Double tolerance; //tolerance of the quadrature problem

    ActorName facili; //facilitator actor's name


    /**
     *  The first response we have received from a child actor. We
     *  save this value until we receive the second response, and
     *  send our client the sum of both values
     */

    ActorName client;
    //the actor which is suppose to receive result from this actor

    String meth; //method to invoke on the receiving client

    /** Creates a new instance of QuadWorker */
    public QuadActor(Double p_douTolerance,ActorName p_anFacili) {
```

```
    tolerance = p_douTolerance;
    facili = p_anFacili;
    meth = "result";

}

public QuadActor(Double p_douTolerance,ActorName p_anClient,
                ActorName p_anFacili) {
    tolerance = p_douTolerance;
    facili = p_anFacili;
    client = p_anClient;
    meth = "result";

}

public QuadActor(ActorName p_anClient,String p_strMeth,
        Double p_douTolerance,ActorName p_anFacili) {
    client = p_anClient;
    facili = p_anFacili;
    meth = p_strMeth;
    numResponses = 0;
    tolerance = p_douTolerance;

}

public void setFaci(ActorName p_anNewFaci){
    facili = p_anNewFaci;
}

/** This method is called to do the quadrature computation
 * a: lower bound
 * b: upper bound
 * tolerance: the error tolerance
 * p_anClient: the actor which is suppose to receive result
 *             from this actor
 * p_strMeth: the name of the method to calculate result in
 *             client actor
 */

public void quadrature(Double a,Double b,ActorName p_anClient,
                                          String p_strMeth){

    double mid = (a.doubleValue()+b.doubleValue())/2;

    Double m = new Double(mid);

    double erroram = error(a,m);
```

```
double errormb = error(m,b);

double tol = tolerance.doubleValue();

if ((erroram > tol)&&(errormb > tol)){
        // need an actor for (a,m) and an actor for (m,b)

    //double percentam = erroram/(erroram + errormb);

    //int ipercentam = (int)percentam;


    try{
        ActorName newChild=
        create("aa.application.AdaptiveQuad.QuadActor",
        p_anClient,p_strMeth,tolerance,facili);

        send(newChild,"quadrature",a,m,newChild,
        p_strMeth);

        send(facili,"resourceAlloc",newChild,
        new Double(erroram));

    }catch (CreateActorException e){
        System.err.println("Exception in QuadActor: "+e);
    }

    try{

        ActorName newChild2=
        create("aa.application.AdaptiveQuad.QuadActor",
        p_anClient,p_strMeth,tolerance,facili);

        send(newChild2,"quadrature",m,b,newChild2,
        p_strMeth);

        send(facili,"resourceAlloc",newChild2,
        new Double(errormb));

    }catch (CreateActorException e){
        System.err.println("Exception in QuadActor: "+e);
    }
}

if ((erroram > tol)&&(errormb < tol)){ //actor for (a,m)
```

```
    try{
        ActorName newChild=
        create("aa.application.AdaptiveQuad.QuadActor",
        p_anClient,p_strMeth,tolerance,facili);

        send(newChild,"quadrature",a,m,newChild,
        p_strMeth);

        send(facili,"resourceAlloc",newChild,
            new Double(erroram));

    }catch (CreateActorException e){
        System.err.println("Exception in QuadActor: "+e);
    }

    result(new Double(trapezoid(m,b)));

}

if ((erroram < tol)&&(errormb > tol)){ //actor for (m,b)

    result(new Double(trapezoid(a,m)));

    try{

        ActorName newChild3=
        create("aa.application.AdaptiveQuad.QuadActor",
        p_anClient,p_strMeth,tolerance,facili);

        send(newChild3,"quadrature",m,b,newChild3,
        p_strMeth);

        send(facili,"resourceAlloc",newChild3,
        new Double(errormb));

    }catch (CreateActorException e){

    }
}

if ((erroram < tol)&&(errormb < tol)){ //no new actors

    result(new Double(trapezoid(a,m)));

    result(new Double(trapezoid(m,b)));

}
```

```
}

/** The function */
public double func(double x){
    //return x*x;
    if (x==0){
        //System.out.println("func 0");
        return 0;
    } else {
        double myFunc =  x*(Math.sin(1/x));
        //System.out.println("func:"+ myFunc);

        return myFunc;
    }

}

/** Calculate the area of a trapezoid */
public double trapezoid(Double a,Double b){
    double aValue = a.doubleValue();
    double bValue = b.doubleValue();
    return (func(aValue)+func(bValue))*(bValue-aValue)/2;
}

/** Calculate the sum of areas of two sub trapezoids*/
public double divideSum(Double a,Double b){
    double aValue = a.doubleValue();
    double bValue = b.doubleValue();
    double midpoint = (aValue+bValue)/2;
    Double dmid = new Double(midpoint);
    return (trapezoid(a,dmid)+trapezoid(dmid,b));
}

public double error(Double a, Double b){
    return Math.abs(trapezoid(a,b)-divideSum(a,b));


}

/** This method is called from another child actor to pass a
 *  partial result. We wait until we have two such results,
 *  add them, and return the result to client.
 *
 */

public void result(Double p_douVal){
    if (numResponses==0){
```

```
            numResponses++;
            partialResponse = p_douVal.doubleValue();
        }
        else {
            //send the answer
            double myResult = p_douVal.doubleValue()+partialResponse;
            send(client,meth,new Double(myResult));

            System.out.flush();
            destroy("Result actor no longer accessible,removing...");
        }
    }

}
```

****

```
/*
 * QuadFacilitator.java
 *
 * Created on June 15, 2005, 9:26 AM
 */

package aa.application.AdaptiveQuad;

import aa.core.FacilitatorActor;
import aa.core.CyberOrg;
import aa.core.ActorName;
import aa.core.Contract;

/**
 *
 * @author  xinghuizhao
 */
public class QuadFacilitator extends FacilitatorActor{

    int num; //keep track of the number of computation actors



    long aveRate;
         //keep track of the average of ticks rate that actors got

    long aveRes; //average of total ticks
```

```
double aveError; //average error so far;

long myRate;
      //ticks rate that will be assigned to new comming actor

long myRes; //ticks that will be assigned to new comming actor

/** Creates a new instance of QuadFacilitator */
public QuadFacilitator(CyberOrg p_cybHost) {
    super(p_cybHost);
    aveRes = 0;
    aveRate = 0;
    aveRes = 0;
    aveError = 0;
    myRate = 0;
    myRes = 0;

}

public void resourceAlloc(ActorName p_anActor,
                          Double p_doubleError){

    if ((aveError==0)||
        (p_doubleError.doubleValue()/aveError<0.1)){

        estimate(p_doubleError.doubleValue());
        try {
            m_cybHost.resourceAlloc(p_anActor, myRate, myRes);
        }catch (Exception e){
            //System.out.println(e);
        }
    }else{
        System.out.println("isolate..");

        estimate(p_doubleError.doubleValue());

        Contract newContract=new Contract(m_cybHost.getTicks()/2,
        (long)(myRate*p_doubleError.doubleValue()/aveError),10);

        ActorName[] actors= new ActorName[1];
        actors[0] = p_anActor;

        try {
        CyberOrg childcyb = m_cybHost.isolate(20, actors,
                newContract);
        send(p_anActor,"setFaci",childcyb.getFacilitator());
```

```
            }catch (Exception e){

            }
        }
    }

    public void estimate(double p_dError){
        if (aveError ==0){
            aveRes = m_cybHost.getTicks()/10;
            aveRate = m_cybHost.getTicksRate()/10;
            aveError = p_dError;
            myRate = aveRate;
            myRes = aveRes;
            num++;

        }else{
            double ratio = p_dError/aveError;
            myRate = (long)(aveRate * ratio);
            myRes = (long)(aveRes * ratio);
            aveRate = (aveRate * num + myRate )/(num+1);
            aveRes = (aveRes * num + myRes )/(num+1);
            aveError = (aveError * num + p_dError )/(num+1);
            num++;
        }


    }

}
```

****

```
/*
 * BootQuad_Res.java
 *
 * Created on June 21, 2005, 11:14 AM
 */

package aa.application.AdaptiveQuad;


import aa.core.Platform;
import aa.core.CyberOrg;
import aa.core.ActorName;
import aa.core.ActorMessage;
```

```
import aa.gui.View;

/**
 *
 * @author  xinghuizhao
 */
public class BootQuad_Res {

    /** Creates a new instance of BootQuad_Res */
    public BootQuad_Res() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        System.out.println(System.currentTimeMillis());

        Integer myPort = new Integer(8070);

        View myView = new View(myPort);

        Platform pPlatform=new Platform(myView,myPort);

        pPlatform.start();

        Object[] nullobja={};

        CyberOrg fibCyber=pPlatform.createCyberOrg(
        Contract.initialContract(),
        "aa.application.AdaptiveQuad.QuadFacilitator", nullobja);

        ActorName facility=fibCyber.getFacilitator();

        ActorName fibonacciActor=null;

        Object[] quadAgs = {new Double(0.0001),facility};


        ActorName quadActor=pPlatform.createActor(facility,
        "aa.application.AdaptiveQuad.QuadActor", quadAgs);


        //adapt for QuadActor (in order to allocate resource)
```

```
        Object[] quadAgs2 = {new Double(0.0001),quadActor,facility};

        ActorName quadActor2=pPlatform.createActor(facility,
        "aa.application.AdaptiveQuad.QuadActor", quadAgs2);

        //end adapt

        Object[] Ags=new Object[4];
        Ags[0] = new Double(0);
        Ags[1] = new Double(1);
        Ags[2] = quadActor2;
        Ags[3] = "result";

        ActorMessage request=new ActorMessage
        (pPlatform.getActorNameOfPlatform(),
        quadActor2, "quadrature",Ags, false);

        pPlatform.sendMessage(request);

    }

}
```