CYBERORGs: A MODEL FOR RESOURCE BOUNDED COMPLEX AGENTS

BY

NADEEM JAMALI

B.C.S., University of Karachi, 1989
M.C.S., Dalhousie University, 1993

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois
Abstract

The ubiquity of networked computing devices combined with recent advances in computer networks technology have created the possibility of harnessing the collective computational power of peer computational resources to carry out very large computations. Agents offer a natural framework for abstracting distributed computations. However, the framework needs to be adapted to explicitly model resources and their ownership in a way that transforms a network of peer-owned resources into a distributed execution environment that can hosts peer-initiated multi-agent computations. Additionally, mechanisms are required for supporting trade in peer-owned resources as well as control of available resources.

This thesis proposes CyberOrgs\textsuperscript{1} as a model for resource bounded multi-agent computations over a network of peer-owned resources. CyberOrgs compose Agents with a model for resource acquisition and control, so that computations carried out by ensembles of agents along with resources committed to their execution are encapsulated inside resource boundaries. Specifically, each cyberorg owns resources and eCash to purchase additional resources with; it manages a concurrent computation being pursued by an ensemble of agents, and supports their execution by purchasing resources for them from other cyberorgs; and, it provides resources to other cyberorgs in exchange for eCash in accordance with contracts negotiated with them.

A transition system has been developed to concretize the operational semantics of CyberOrgs, and some properties of the model have been examined. A prototype implementation has been developed as an Actor program which offers programming constructs to support acquisition and control of resources, while allowing a programmer to keep resource and functional concerns of an application separate. The expressive power of these constructs is illustrated using examples.

\textsuperscript{1}The model will henceforth be referred to as CyberOrgs, and the entities will be referred to as Cyberorgs.
To my parents and teachers.
Acknowledgments

I would like to thank Professor Gul Agha for advising me toward my Ph.D. Not only did he supervise my dissertation research, through the course of these years, he has been very influential in shaping my view of scientific scholarship, its role in the state of human society, and the importance of scientists taking their responsibility seriously. His encouragement was particularly critical at times when I wavered in my commitment to pursuing a career in academia. Over the years, Prof. Agha and his wife, Professor Jennifer Cole, have both been a tremendous support at times of academic and personal challenges.

I am immensely grateful to my dissertation committee for taking an active interest in the quality of my work. In particular, I would like to thank Professor Les Gasser for his generous advice toward improving my thesis and for sharing important insights from his wide experience in the area of Multi-agent Systems. Professor Jean-Pierre Briot’s insistence on highlighting the delta contribution of my work served as a very valuable principle which I expect to continue to benefit from. Professor Vikram Adve’s input provided a unique perspective required for putting my work in the broader context of software systems. Last, but certainly not the least, I am thankful to Professor Mehdi Harandi, for not only his generous advice on the thesis, but also for being a very encouraging presence in the Department of Computer Science as the Director of Graduate Studies, and for taking a personal interest in my success.

Sincere thanks go to Eastman Kodak Company, for supporting my research through an award of the Kodak Fellowship. I am also grateful for research assistantships received through a DoD MURI grant (AFOSR contract F49620-97-1-03821) and DARPA’s TASK project.

The University of Illinois provided a world class environment for my academic and intellectual growth. Not only did it present me with the opportunity to pursue interests in my research area, it also afforded me access to resources such as the marvelous library system, enabling me to take
on broader academic and intellectual pursuits. Additionally, University of Illinois offered ample opportunity to learn simply from being a part of the great institution.

The Department of Computer Science with its legendary place in the history of the discipline could not have been a more stimulating environment for carrying out quality research. The supportive role of the Department in the education of graduate students was perhaps best personified in Professor Michael Faiman, who as the Director of Graduate Studies, made personal efforts on my behalf a number of times, for which I am indebted. Barb Cicone’s thoroughness was always a source of comfort.

The Open Systems Laboratory provided an environment which had the best mix of challenge and support. With a strong culture of respect for independent thought as well as vigorous debate, OSL provided me the freedom to pursue my research goals while holding me to the highest standards. For this I remain thankful to all past and present members of the lab. I am particularly thankful to Mark Astley, Myeong-Wuk Jang, Wooyoung Kim, Shangping Ren, Dan Sturman, Prasanna Thati, Carlos Varela, Nalini Venkatasubramanian and Reza Ziaei, for taking interest in my work at different stages and being good friends. Joonkyoo Yoo must be acknowledged for helping all those who came into contact with him have a different perspective on life. I would also like to thank departmental Academic Assistants – Andrea Whitesell, Bonnie Howard, and Anda Ohlsson – who were always there and eager to help.

I am grateful to a number of friends simply for being reliable friends most of the time. These include Stephanie Atkins, Altaf Memon, Asif Shah, Nadir Sherali, Rahman Soomro, Ramesh Tharwani, Ishaq Tumio, and Khalid Zaman. I should also mention the number of Sindhi cyber-communities I was part of, which provided necessary and sometimes not so necessary distraction.

Finally, I would like to thank my family for never failing in their encouragement. My father, Maula Bux Jamali, who is no longer with us today, taught me that it was worth pursuing things I believed in. My mother, Khurshid Bano, was always there when I needed the type of support that only a mother can offer. My siblings – Naveed, Sophia, and Owais – have always looked up to me as an elder brother, which although intimidating at times, has been a setter of bars worth crossing. The pride felt by the extended family in Nawabshah, Sindh, has been a reliable source of encouragement.
## Table of Contents

**List of Figures** ........................................... x

**Chapter 1  Introduction** .................................. 1
  1.1 Key Concerns ........................................... 2
      1.1.1 Coordination .................................... 2
      1.1.2 Resource Control ................................. 3
      1.1.3 Resource Acquisition ............................. 5
  1.2 Outline ............................................... 6

**Chapter 2  Related Work** ................................. 7
  2.1 Mobile Agents ......................................... 8
      2.1.1 Actor Model Research ............................. 9
  2.2 Coordination ......................................... 12
  2.3 Resource Control ..................................... 13
      2.3.1 Quantum ....................................... 14
  2.4 Grid Computing ....................................... 14
  2.5 Security ............................................. 14

**Chapter 3  A Model of Resources** ....................... 16
  3.1 Approach ............................................. 16
      3.1.1 Resource Acquisition and Control .............. 17
      3.1.2 Separation of Concerns .......................... 18
  3.2 Properties of Resources as Tradable Objects ......... 18
  3.3 Organization of Resources ............................ 19
  3.4 Mobility of Computations .............................. 20
      3.4.1 Patterns of Mobility ............................. 21

**Chapter 4  CyberOrgs** .................................. 22
  4.1 Control Hierarchy .................................... 22
  4.2 Cyberorg Views ....................................... 23
      4.2.1 Hierarchy of Cyberorgs .......................... 23
      4.2.2 Set of Cyberorgs ................................ 25
      4.2.3 Transparent Middleware .......................... 26
  4.3 Trading in Resources .................................. 27
      4.3.1 Dynamic Pricing ................................ 27
  4.4 Problem Classes ..................................... 27
      4.4.1 Remote Execution ................................. 27
4.4.2 Local Control .............................................. 28
4.4.3 Execution in Open System ................................. 28
4.5 Example: Insurance Company ................................. 30

Chapter 5 Operational Semantics .................................. 31
5.1 Actors ......................................................... 31
  5.1.1 Actor Syntax and Semantics ............................... 32
5.2 Instantiating CyberOrgs with Actors ....................... 35
  5.2.1 Progress .................................................. 36
  5.2.2 CyberOrg Primitives ...................................... 38
5.3 Design Decisions .............................................. 42
  5.3.1 Representation of Actors .................................. 42
  5.3.2 One Tick at a Time ...................................... 44
  5.3.3 CyberOrg Structure ..................................... 45
  5.3.4 CyberOrg Primitives .................................... 46
  5.3.5 Expiration of Ticks .................................... 47
  5.3.6 Ownership .............................................. 47
5.4 Properties .................................................... 48
5.5 Discussion ................................................... 49
  5.5.1 Typed Ticks .............................................. 49
  5.5.2 Resource Consumption Patterns ...................... 50

Chapter 6 Implementation ........................................... 51
6.1 System Design ............................................... 51
  6.1.1 Artifacts ................................................ 51
  6.1.2 Control .................................................. 52
  6.1.3 Strategies .............................................. 55
  6.1.4 User Interface ......................................... 55
6.2 Implementing an Application ................................ 56
  6.2.1 Cyberorgs .............................................. 56
  6.2.2 Application Actors .................................... 58
  6.2.3 Negotiators ............................................ 58
  6.2.4 Strategies ............................................. 60
6.3 Implementation Details ...................................... 60
  6.3.1 CyberOrgs .............................................. 60
  6.3.2 Application Actors .................................... 62
  6.3.3 Contracts .............................................. 62
  6.3.4 Negotiation ............................................ 62
  6.3.5 Tick Propagation ..................................... 64
6.4 Discussion ................................................... 65
  6.4.1 Atomicity of Methods ................................. 66
  6.4.2 Optimizations ......................................... 66
  6.4.3 Overhead .............................................. 68
List of Figures

1.1 Reactive Control: When purging threshold is n, the system purges an agent if it is observed to create n new agents in a window of 15 opportunities to do so. Each agent is born with a probability of creating a new agent on opportunity, which is randomly picked from a uniform distribution. The system begins with 2 agents in each case. The growth in number of agents turns out to be exponential for purging thresholds greater than 1. .............................................. 4

4.1 Hierarchy of Cyberorgs: Contracts stipulating availability of resources from cyberorg to cyberorg organize a system of cyberorgs as a tree. The delivery of resources is hence hierarchical. .......................................................... 24

4.2 Set of Cyberorgs: Cyberorgs may be viewed as a set of peers with independent control over their resources. Cyberorgs trade in resources with other cyberorgs independently, irrespective of cyberorgs currently “hosting” them. ................. 25

4.3 Transparent Middleware: A system of cyberorgs may be viewed as a transparent middleware supporting resource requirements of an application. This is possible because CyberOrgs separate resource and functional concerns of an application. . . . . . 26

4.4 Insurance Company: An insurance company’s functions organized using cyberorgs. Agents created to handle newly received claims are encapsulated in cyberorgs with resources dedicated to handling the claims. In each cyberorg dedicated to a claim, new cyberorgs are created to handle specific aspects of the claim, which migrate to a remote cyberorg for research and return with results. ......................... 29

5.1 Actors: Actors encapsulate a thread and state. The interface is comprised of public methods which operate on the state. Actors may create other actors and may send asynchronous messages to other actors. .............................. 33

5.2 Cyberorg Isolation: A cyberorg encapsulates a number of its actors, messages and an amount of its eCash as a new cyberorg with independent control. The isolating cyberorg becomes the host of the new cyberorg and has a contract by which the new cyberorg will be served. ..................................................... 39

5.3 Cyberorg Assimilation: A cyberorg assimilates into its host cyberorg by relinquishing independent control of its eCash, actors and messages. The contents of the assimilating cyberorg become contents of the cyberorg originally hosting it. ........ 39

5.4 Cyberorg Migration: A cyberorg migrates from its current host cyberorg to another cyberorg. This happens after a prospective host has been found and a contract has already been negotiated with it. ........................................... 42
5.5 Remote Message: A message created by an actor in a cyberorg directed at an actor located inside another cyberorg is encapsulated into a new cyberorg. The new cyberorg migrates to the remote cyberorg containing the intended recipient actor, and assimilates into the host cyberorg on arrival, breaking its encapsulation and turning the message into a local messages which can be delivered.

6.1 checkForTriggers method: Method which checks for thresholds to determine whether it will be advantageous to carry out of the primitive cyberorg operations. These requests are inserted into a queue called chores, which is dequeued by the system later.

6.2 initiateMigrateSequence method: Method which communicates with the yellow page service to find a potential host cyberorg, and once found, sends a request for resources to the cyberorg to begin negotiation for a contract.

6.3 relocateToServer method: Method to be called by an application actor to explicitly request a migration of the containing cyberorg to a particular cyberorg. This request results in a resource request being sent to the prospective host, which would initiated a negotiation.

6.4 ClaimActor class: An actor that handles an individual claim. It creates actors for handling different aspects of the claim, and requests its containing cyberorg to isolate each of these actors into dedicated cyberorgs. The new cyberorgs migrate to server cyberorgs with specialized resources to handle specific aspects of the claim. Eventually, these cyberorgs would return with actors holding results.

6.5 Negotiator class: Class from which server and client negotiator classes subclass. These actors act on behalf of cyberorgs they are contained in, to negotiate contracts with other cyberorgs. A negotiator actor is created in response to a resRequest message received by a cyberorg.
Chapter 1

Introduction

A distributed computation may be viewed as an attempt to achieve desired properties of states of a network of computers at instants or during time intervals in the future. Examples of these properties include states of availability of solutions to a problem, states of well-behavedness of a distributed operating system, and so on. To reach or maintain such states, computations should execute in environments which allow them to reasonably pursue their goals. A property of such an environment is its predictability to the control mechanisms of computations.

A natural way of modeling distributed computations in an open system is by using agents. We define agents as active objects with individual or group goals, and resources dedicated to achieving those goals. Goals of an agent or a system of agents (a multi-agent system) may be represented as desired states in time and space. Resources available to an agent may be diverse and distributed.

This thesis develops mechanisms for trading in resources and reification of resource control for supporting multi-agent computations in an open system, and constructs a model for reasoning about these mechanisms. There are four specific contributions of this research: a framework for organizing resources as a market, and for reifying control of available resource; an operational semantics for reasoning about the framework; programming abstractions and constructs for supporting implementation of systems based on the framework; and a prototype implementation to illustrate the expressivity of the framework.
1.1 Key Concerns

Although networking makes it theoretically possible for individual multi-agent computations to use virtually limitless computational resources, freeing computations from being bounded by resources which are statically dedicated to their execution, large distributed systems offer a very complex execution environment for carrying out computations. Not only is it not entirely clear how a multi-agent computation might to be mapped to the environment, there is no obvious basis for sharing of remote computational resources which are typically independently owned. There are three specific concerns: coordination, control, and acquisition of resources.

1.1.1 Coordination

A fundamental concern in mapping a computation to a distributed system - or to a concurrent paradigm in general - is to coordinate the execution of sub-computations. Coordination is the task of reducing uncertainties which make it difficult for individual components of a collaborative computation to make control decisions. Control decisions are decisions each computing entity must make about what to do next, so as to make the greatest advance toward the shared goal [21].

There are multiple sources of uncertainty in a computational environment. A computation’s resource requirements may change in application dependent ways (for example, because requirements at any instant may be determined by some function of partial results). An application independent source of uncertainty for a computation emanates from other computations competing for available resources, in effect threatening an individual computation’s prospects of receiving resources that it requires. In an open system [27] where computations may enter or exit the system at any time, this is a typical scenario. In other words, resource dependencies exist between computations which lead to uncertainty of resource availability.

Where multi-agent systems solve problems by letting agents compete or collaborate in a problem space, when computational resources are bounded, agents will invariably compete in the resource space as well. Unrestricted competition for resources between agents collaborating to achieve a shared goal may hamper progress toward the shared goal [21]. Coordinating resource access by agents is hence critical to reducing uncertainty and enabling agents to make control decisions for best global performance [32].
1.1.2 Resource Control

In a bounded resource environment, if a computation can launch other computations as in a multi-agent system, it is difficult to control resource consumption reactively. If an erroneous or malicious agent begins creating other agents with similar characteristics, and if the only mechanism employed for identifying such agents is observation of their own threatening behavior, the rate of growth in the number of agents can be shown to be exponential. Intuitively, this means that irrespective of how conservatively the system tries to purge misbehaving agents, so long as the mechanism relies solely on the observation of suspicious activity, by the time the system reacts, it may be already too late: other agents have potentially been created about whose behavior the system will know nothing until it has observed them individually. The consequence of such growth in the number of agents will depend on how agents are scheduled. If each agent has its own thread, and is scheduled by the operating system, the system will soon be overloaded, and if a limited number of threads are permissible, the system will run out of threads. Until such a time, well-behaved computations will suffer exponential delays in their completion.

A back of the envelope calculation illustrates the difficulty. Consider a scheduler that schedules agents for fixed time slices in a round robin fashion. If the probability of an agent creating another agent when given an opportunity is $p$, and the system purges an agent when it observes its behavior to exhibit a creation probability of $t$, if we begin with $n$ such agents, at the end of the end of the $n^{th}$ cycle of the scheduler, the number agents is $(n(1 - p/t))^t$, which represents an exponential growth.

As also illustrated in Figure 1.1 showing simulation results, even a very aggressive reactive mechanism, which purges agents for creating even a moderate number of other agents, is bogged down by the exponential growth in the number of agents.

An effective mechanism for controlling such behavior would require tracking groups of agents. In other words, at the time of purging an agent, if there were a way of identifying other agents whose creation is rooted at the purged agent, all of them could be purged together. However, because of the exponential growth described above, a book-keeping solution of this problem is impractical. Specifically, the cost of maintaining information about which agents are created by which other agents – to be used for purging all agents which were created (directly or indirectly) by an agent being purged – also grows exponentially.
Figure 1.1: Reactive Control: When purging threshold is $n$, the system purges an agent if it is observed to create $n$ new agents in a window of 15 opportunities to do so. Each agent is born with a probability of creating a new agent on opportunity, which is randomly picked from a uniform distribution. The system begins with 2 agents in each case. The growth in number of agents turns out to be exponential for purging thresholds greater than 1.
An alternate approach to control is by bounding resource consumption at the outset, and limiting resources available to a computation and all sub-computations originating from it. In this approach, each agent would receive a resource consumption allowance, which it could utilize or give a part of to other agents.

There is an added control overhead involved in an agent’s distribution of parts of its allowance to other agents; however, cost of the overhead is isolated to the computation creating new agents. Another problem is deciding how much resource to allocate to a sub-computation. Unless resource requirements of computations can be accurately estimated, obtaining needed resources may involve communication between layers of the computation, which too contribute to overheads. However, once distribution of resources to various computations at any time is established, agents may make resource usage decisions as part of their control decisions for coordination.

1.1.3 Resource Acquisition

An obvious problem arises when multiple problems with conflicting (or competing) goals can compete for an execution environment. Because goals of different problems must be conjuncted to obtain the collective goal, conflicting goals result in failure. This scenario is more likely when problems may have temporal constraints as parts of their requirements.\footnote{We contend that restricting temporal constraints for only the most time-sensitive (e.g., real-time) applications is too restrictive. There is almost always some expectation of a reasonably timely completion, and such expectations must be represented in the form of (however flexible) temporal constraints for better analysis.}

This problem can be addressed naturally if we consider ownership properties of entities in the system. Note that ownership properties are typically ignored in modeling of computations because they are assumed to be simple and statically determined. For example, when multiple problems are solved on the same computer, the problems are multiplexed in time by variants of a prioritization scheme. However, this simple approach proves inadequate when we allow problem specifications to include temporal constraints.

We assume that all entities in a distributed system have owners at any instant of time who may allow access to the entity. For example, each resource (hardware, software, or data; physical or logical) is owned, and any computation on the node must be subject to the owners’ permission; similarly, each problem and tasks pursuing its solution in the system also have owners. Ownership
defines access and responsibility boundaries around each entity; access to each entity must be allowed by its owner, and an entity’s owner is responsible for all its actions.\footnote{The owner here is not in the sense of an individual human who owns the entity; ownership is in terms of the interest the entity is serving. Hence, multiple tasks initiated by the same person may be thought of as having different owners because they are solving different problems.}

Once ownership rights are incorporated into the model, not only do we have a basis for handling multiple computations with conflicting goals, we may also introduce a mechanism for trade in these resources. Goal specifications of computations may now include allowances in the form of some currency for acquiring resources. Computations may no longer make demands from resources that are not owned by the computation’s owner.

### 1.2 Outline

In the chapters to follow, the CyberOrg model will be presented. Chapter 2 discusses the existing body of work in a number of areas related to the research being presented here. Chapter 3 discusses important characteristics of computational resources which motivate the model. Chapter 4 introduces CyberOrgs, and informally describes their structure, multiple ways in which a system of cyberorgs may be viewed, identifies specific classes of problems which would benefit from CyberOrgs’ facilities, and ends with an example. Chapter 5 presents an operational semantics of CyberOrgs in the form of a transition system, and discusses some simple properties held by CyberOrgs. Chapter 6 described an implementation of Cyberorgs as an Actor program, built using the Actor Foundry library of Java classes [36]. Chapter 7 highlights possible implications of this work and identifies some areas of future work. Finally, Chapter 8 concludes the thesis.
Chapter 2

Related Work

A number of related problems were addressed pre-Internet in the context of concurrency, distributed artificial intelligence, and multi-agent systems. The Actor model [2] offered a promising abstraction for implementing object-oriented distributed systems. The model offered insight into how concurrent object-oriented computations could be viewed as open systems [27].

Where the emergence of Java and the resulting ability to write portable software fueled a mushrooming of mobile agent systems, a set of problems emerged which hindered wider deployment of systems based on the paradigm. Ironically, Java’s approach to portability presents part of the challenge: in order to make software portable, a design decision was made to conceal low level resource information from programmers. Although this is a reasonable approach for some purposes, it neglects the fact that resource control is more – not less – important in the presence of mobile computations. Specifically, when mobility of computation is permitted, it is important to control the resources that the computation consumes on the machine where it is hosted.

Although research in the real-time systems community has long focused on the problem of guaranteed temporal behaviors of computations, temporal constraints have traditionally been a less stringent requirement for general purpose computations. This may be acceptable when the closed-world assumption holds, and reasonable expectations can be made about timeliness of computations based on the local state. With the networking of computers and potential for mobility of computations, the closed-world assumption no longer automatically holds; as a result, timeliness of computations can no longer be guaranteed by separately considering factors such as processor and network performance. Consider some ways in which performance is difficult to predict in open systems. For example, such systems may suffer from increased vulnerability to disruption. Viruses
and denial of service (DoS) attacks dramatically illustrate what it means for the closed-world assumption not to hold. However, in the presence of mobile computations, this is not simply a matter of security, but that of resource control. In fact, viruses and denial of service attacks are seen as security breaches precisely because the models underlying stand-alone computers do not address openness. Consequently, these models are inadequate for reasoning about mobile computations.

As evidenced by the proposed addition of Isolation and Resource Management in Java [11], which offer resource control abstractions and mechanisms to programmers, Java's initial approach for preventing programmers from breaching the system's abstractions was more restrictive than was necessary. Specifically, although it would not be safe for a program to look at (say) data representations underneath the virtual machine's abstractions, it is necessary for the amounts of resources used for the representations to be revealed. This advance was in part influenced by earlier work on separating resource concerns from application concerns for CyberOrgs [31].

2.1 Mobile Agents

HAL [34] was one of the earliest implementations of mobile agents, which provided migration of agents (actors) as part of an efficient implementation of the Actor [2] model of computation. Agent Tcl [25] was perhaps the most popular early mobile agents project. Implemented in the Tcl scripting language, Agent Tcl specifically addressed issues in implementing mobile agents in an environment with intermittent connections between computers. It reduced migration to a single instruction, provided transparent communication among agents (hiding all transmission details), and allowed a simple scripting language to be used as the main agent communication language.

More recently, mobile agent systems have been typically built using Java for portability. However, because of inadequate support for resource control in Java, some implementations have customized the Java virtual machine, compromising Java's portability. Nomads [58] is one such example. JRes [12] offered an early variant of the Java virtual machine that offered low-level for systems developed in Java.

The dominant theme in mobile agents research has historically been process migration over a network. These processes may be called agents only by a very loose definition of agents. For

---

1 Agent Tcl too has been reimplemented in Java and is now called D'Agents.
example, Nomads uses a separate instance of its modified Java virtual machine for every agent. Although it makes it possible to offer strong migration, the overhead is significant [26].

Resource control has emerged as an important concern in the mobile agents community. Important projects addressing resource control are JSeal2 [62], KafEOS [38], besides D'Agents and Nomads mentioned above. JSeal2's precursor JavaSeal [63] was influenced by early work in the development of CyberOrgs, on separation of concerns between application and resource concerns for resource-bounded agent ensembles [31]

The Mobile Agent Facility Specification by the Object Management Group [30] has made a case for standardizing areas of mobile agent technology to promote interoperability. These include agent management, transfer, naming (agent as well as agent system), agent system types and location syntax.

2.1.1 Actor Model Research

Actors [2, 3] provide a formal model for building and representing the behavior of concurrent objects and thus serve as a foundation for concurrent object-oriented programming.

Actors are autonomous, interacting computing elements, which encapsulate a behavior (data and procedure) as well as a process. Different actors carry out their actions asynchronously and communicate with each other by sending messages. The basic mechanism for communication is also asynchronous and buffered; however, other forms of message passing can be defined in the context of the model. Finally, actors may be dynamically created and reconfigured, which provides considerable flexibility in organizing concurrent activity.

Actors are a model for specifying coordination in a distributed system. Because the internal behavior of an actor is encapsulated and cannot be observed directly, the Actor model supports heterogeneous, variable grained objects. Specifically, the behavior of individual actors may be defined using any programming language.

Actors provide a natural extension of the object-oriented paradigm to concurrent and distributed computation. They support encapsulation, description as behavior templates, and re-usability via libraries accessed using message-passing protocols. The locality properties of actors guarantee that

\footnote{Being able to migrate instantaneously, as opposed to weak migration which migrates an agent at the completion of ongoing method execution.}
changes of representation and elaborations can be made independent of the interaction with, and
behavior of, other actors. Thus actors can support local instrumentation and monitoring which
provide important tools for analysis and debugging.

Reflection

Models of reflection enable interaction of higher level operations, such as real-time constraints, and
lower level information about the execution environment, such as load distribution over a group of
processors, available network bandwidth, etc.

Because the Actor model allows the state of the computation to be modeled directly, the com-
putation environment called the meta-level architecture can be represented at an appropriate level
of abstraction using the same base language [60]. Specifically, this allows use of reflection enabling
an agent to have a continuous interaction with the environment to determine available resources
and relate it to its own state to provide evolving resource consumption strategies.

In Rosette [59], a commercially developed object-oriented implementation of an Actor archi-
tecture, the architecture has an interface layer and a system environment. The interface layer
provides mechanisms for monitoring and control of applications, where the system environment
contains actor communities which implement resource management policies, providing monitoring,
debugging, resource management, system simulation, and compilation/transformation facilities.

Interaction Protocols

An interaction policy may be expressed in terms of the interfaces of actors and implemented by using
appropriate protocols to coordinate actors. A protocol imposes a certain role on each participating
actor. In essence it mediates the interactions between actors to ensure that each relevant actor
implements its end of the interaction policy.

Sturman and Agha have developed a language for describing and implementing interaction poli-
cies [56, 57]; using this language, a protocol abstraction may be instantiated by specifying a
particular group of actors and other initialization parameters. The runtime system must then
support specific forms of reflection, which are sufficiently powerful to enable dynamic modification
of the mail system and to store and retrieve actor states, or other parts of the meta-architecture.
Synchronizers

Synchronizers [20] are a language abstraction that expresses two types of synchronization constraints, simplifying the task of distributed programming. The first type imposes precedence constraints on otherwise asynchronous events at different actors, and the other requires such events to be atomic (loosely speaking, to co-occur). Because synchronizers may be superimposed, and may be dynamically added or removed, implementing such a system efficiently proves to be a fairly challenging but is nevertheless feasible.

RTsynchronizers

RTsynchronizers [50] offer one way of implementing real-time constraints using an abstraction similar to that for the declarative coordination constraints discussed earlier. RTsynchronizers are objects that enforce real-time constraints by constraining whether or not messages of a certain type can be delivered to an actor at a certain point in time.

ActorSpaces

The ActorSpace model allows an abstract specification of a group of actors [7]. An actorspace associates an actor with specific attributes; the sender of a message specifies a destination pattern which is pattern-matched against the attributes of actors in the actorspace. The model may also be seen as providing a distributed version of the blackboard [18] system for broadcast communication. A simple analogy with set theory illustrates the difference between naming in actors and actorspaces. A set may be defined by enumerating its elements, or by specifying a characteristic function which defines a subset in a domain. The first method is analogous to actor communication (where an explicit collection of mail addresses of actors must be specified), whereas the second method corresponds to actorspace communication. Of course, in conventional mathematics the two ways of characterizing sets are equivalent since the properties of mathematical objects are static; by contrast, actors may dynamically change their attributes. Actorspace provides a transparent way of managing groups of actors. It generalizes the notion of ports in process calculi, where object identity is also not uniquely defined, but pattern are degenerate.

In multi-agent systems, it is important to be able to access new services that become available
and to know when existing servers no longer exist. This necessitates a pattern-based naming scheme that identifies agents as being members of groups and allows communication with agents that are not individually known. These group identifiers can also be used in defining protocols.

2.2 Coordination

Coordination has long been seen as a key concern in distributed computing [6]. Furthermore, it has also been argued that computation and coordination are separate and orthogonal dimensions of all useful computing [23]; consequently coordination needs to be addressed explicitly.

When a computation is distributed into semi-autonomous subcomputations collectively solving a problem, a degree of uncertainty is inevitably introduced. In the context of multi-agent systems, when the decision of an agent about which action to take next depends on actions taken by other agents, coordination between the agents is required to achieve optimal results [21].

Computations sharing an execution space inevitably compete for the resources in that space. There may be both logical or resource dependencies [21] between agents, with the resource dependencies sometimes leading to logical dependencies. Consequently, it is often essential to coordinate resource use to achieve coherent behavior [32].

Three important ways in which coordination has been achieved have been by using organizational structures, meta-level information exchange, and multi-agent planning [32]. Organizational structures rely on long-term information about agents instead of expensive explicit communication. For instance, agents may use models of behaviors of other agents in order to coordinate [54]. A more centralized alternative may be something akin to “social laws” adopted by a society of agents, as in [53]. Meta-level information exchange, such as in Partial Global Planning [17], relies on exchange of high-level information between agents about their medium-term plans to enable the agents to adapt. Multi-agent planning is the most computation and communication intensive mechanism for coordination: such planning attempts to avoid inconsistent or conflicting actions by agents by specifically focusing on sharing of scarce resources and exchanging detailed short-term information between the agents. A multi-agent plan may be developed in a centralized [24] or distributed [10, 51] manner, to which all agents must remain committed to avoid conflicts [52]. There are specific costs associated with maintaining organizational structure for coordination [42],
most notably the cost of communication required to achieve coordination, and the cost of failure to coordinate because of inability of the coordination mechanism to keep up with changes.

Real-Time Task Scheduling

Emergence of Internet has brought renewed attention to timeliness and resource consumption. An important reason for this is that unlike computations happening on dedicated or centrally managed computers or networks of computers, parts of computations involving the Internet occur in an execution space with non-local control. As a result, timeliness of computations is much less tractable than before [8].

Task scheduling is an important aspect of the coordination problem: the order of execution of tasks must satisfy logical as well as resource constraints. Of particular relevance to this thesis is the work on planning under scarce resources [15, 29]. Specifically, Design-to-Criteria scheduling [64] uses reasoning about the quality, cost, duration, and uncertainty trade-offs of different courses of action to construct schedules with desired goals. This approach has been applied in the BIG system [39] for resource-bounded information gathering on the World Wide Web. To alternative courses of action, the TAEMS task modeling language [16] is used which organizes tasks hierarchically, maintaining information about cost, quality and duration of execution for each primitive action, as well as whether and how the tasks or primitive actions may interact.

2.3 Resource Control

Ether [35] was the first language to address explicit allocation of resource in concurrent systems. Sponsors were assigned to processes to support their computations. This idea was later incorporated in the Actor language Acore [44]. Sponsor actors accompanied computation requests, and they carried ticks that could be used in processing a request. Using a similar scheme, in Telescript [65], processes were awarded funds in terms of teleclicks which they were supposed to use to accomplish their results.
2.3.1 Quantum

The Quantum [45] framework is the most relevant to our work on Cyberorgs. Motivated to serve the need to manage finite resources shared by multiple computations, in Quantum, computations require energy to execute. Computation tasks are contained in groups which also serve as tanks of energy. Groups are hierarchical like cyberorgs, so that a group may create subgroups with its subcomputations. When a group’s computations terminate, its energy is absorbed into the energy of its parent group; when it has exhausted its energy, it may receive more energy from its parent. Although the original formulation of Quantum did not support migration over multiple hosts, it has since been extended [46] to handle management of distributed and multi-type resources, which does address migration in a limited manner.

2.4 Grid Computing

The idea of harnessing the power of idle computational resources connected by networks is not new. Condor [40], developed in 1987, was one of the early systems that enabled pooling of Unix based machines in a network for solving large problems. One part of the system managed jobs submitted to the system, and another managed resources on which those jobs may be executed.

With the emergence of Internet and relatively faster improvements in performance of networks than computers, there has been a resurgence of interest in Grid Computing with many generously funded projects around the globe (e.g., [47, 28, 19, 1]).

Many advances in the field of Grid Computing are relevant and complimentary to the work described in this thesis. The Globus Project [19], for example, is building a toolkit for sharing and accessing large and possibly heterogeneous resources over networks. The Resource Specification Language [13] developed for communicating resource requirements in the context of Globus, as well as the Service Negotiation and Acquisition Protocol [14] for managing resources in a distributed multi-owned network, may be adopted in the design of Cyberorg systems.

2.5 Security

Deployment of mobile agents in an open system poses significant security concerns [61].
Traditional approaches to handling security have enforced user-based access restrictions on computational resources; reasonable assumptions could be made to sufficiently narrow down the security threats to manageable levels. Because mobile agent systems require greater flexibility and dynamicity in resource access, these assumptions no longer hold [9]. One approach is to limit agents to executing in clearly defined trust domains, but it obviously restricts mobility. Alternatively, reputation servers [37] could be deployed to track actions of agents. However, all these approaches work only under the closed-world assumption.

Protection against a set of agents or hosts collectively causing undesirable behavior in an open system is a challenging problem. Emergent behaviors may be controlled by using preventive mechanisms. These mechanisms may rely on linguistic support for precluding undesirable patterns of behavior, or they may attempt to detect imminent threat and take steps to prevent it. Each approach has its problems: where protection for all conceivable types of threats cannot be incorporated in the language, detecting threat or imminent threat is also difficult. How group behavior emerges from behaviors of constituent entities, is not well understood [22]. Even passive messages can flood a network [43]. Certain group behaviors, however, such as resource consumption, are amenable to analysis at higher levels of abstraction. In other words, resource consumption is an example of a group behavior which is possible to track and control in a scalable manner by monitoring the resources and managing agents’ access to them.
Chapter 3

A Model of Resources

Multiagent systems have evolved as a promising framework for distributed computing. Multiagent systems are systems of autonomous mobile agents pursuing shared goals. Goals of multiagent systems typically involve spatial and functional components. Agents navigate their way through distributed systems, searching for environments suited for their execution.

Resources needed by a set of agents will typically be in the form of access rights. A resource may be needed for a certain amount of time and at a (possibly varying) rate of availability. For example, an agent may require 10 seconds of processor time at the rate of 1/2 second every second. In a market based approach, the agent would negotiate with the processor's owner on the terms of usage, and pay for the resource in units of a mutually agreed currency.

An interesting implication of the ability to secure rates of availability of resources is that it creates a reasonable expectation of solving problems in a timely manner. Specifically, it allows us to solve problems that have temporal constraints in addition to the spatial and functional requirements that form typical problems. This brings us closer to the most general set of problems solvable using distributed systems. Furthermore, the ability to secure access rights in exchange for units of currency has an additional implication of allowing the system to interact with the “real world”. Electronic commerce over Internet would be an instance of such interaction.

3.1 Approach

To address multi-agent interactions in a resource space, we draw inspiration from two sources: populations of biological organisms in nature, and commercial organizations in human societies.
Organisms in an ecology compete for nutritional resources; however, once the resources have been acquired, they are shared between the organism's various organs in a coordinated manner. Similarly, where commercial organizations compete with each other for financial resources, decisions about allocation of resources between various parts of the organization may have a centralized component. In general, such systems may be seen as hierarchies where there is competition as well as centralized decision making at each level of the hierarchy. We organize multi-agent systems in a similar manner.

Our approach is to encapsulate resources and the ability to secure new resources (represented as eCash) along with a concurrent computation to create funded resource-bounded entities called CyberOrgs. Furthermore, we do this while maintaining a separation between application and resource concerns.

3.1.1 Resource Acquisition and Control

We view a large peer-owned network of computers such as the Internet as a set of resources with rights of ownership assigned to them.

A biological organism is typically a collection of co-located organs encapsulated inside a wrapper, collectively bound by the set of resources available to them. Each organ has the biological analog of a thread of control, and these organs interact under tighter constraints, as opposed to the type of looser constraints which determine how one organism interacts with another. For example, even though the legs of an ant can move simultaneously and independently of each other, the goal of self-preservation or preservation of the colony is hard-wired in the ant through evolution. The resources needed for this pursuit are also secured at the level of an ant. The way in which individual legs act is determined by how the resources available to the ant are distributed among its organs, including the legs, and the ways in which the legs are constrained to behave with respect to the rest of the organism. A significant implication of co-location is that the organs share, and are known to share, a common external environment in which they operate. This allows enforcement of interaction constraints that can be fine-tuned to very precise details.

We introduce CyberOrgs (Cyber Organisms) as a model for complex distributed computation inspired by this view of organisms in the real world.
3.1.2 Separation of Concerns

Cyberorgs enable a separation of concerns between the application’s functionality and its resource needs. Specifically, a cyberorg is responsible for facilitating its managed computation by securing resources it requires. Resource requirements are specified by the programmer separately from the functional part of the application. The cyberorg monitors the environment and the computation’s progress until it determines that migration may be beneficial. At this point, it begins scouting for potential host cyberorgs, and negotiates with them to decide whether to migrate and, if so, where.

3.2 Properties of Resources as Tradable Objects

Resources needed for solving computational problem over networked computers have important characteristics which have a bearing on how they must be traded, and how their use must be controlled.

Types of Resources There are various types of resources, depending on their functions, their specifications, and their locations. Types of resources may be organized as a hierarchy so that they are divided by function at the highest level, and by other attributes at lower levels.

Expiration Computational resources have a spatiotemporal existence. In other words, units of these resources are available for use at certain instances, before which they are not ready for use, and after which they no longer exist - or expire. For example, a unit of processor time is available at a certain instance; if not used at that instant, it no longer exists. The same is true for other types of resources as well. For example, an amount of available memory or network bandwidth may be used at the instant of availability; if unused, the availability at the particular instant cannot be exploited at a later instant.

Exchangeability Multiple instances of a particular type of resource may be interchangeable. For examples, two units of a memory space of the same size, at the same machine, available at the same instant, may be used in place of each other.

Patterns of Requirements Resources may be needed at absolute instances or at instances relative to local or non-local events. Resources may be needed independently of each other, or
they may be needed along with other resources of the same or different types.

3.3 Organization of Resources

Hierarchy of Ownership Although ownership of objects in space may be seen as a flat relationship between objects and their owners, ownership of objects in space and time should be represented as a hierarchy. This is because any transfer of ownership is for an amount of time and it lapses at the end of that time interval. This makes it important to keep track of whom the ownership of an object must return to at the end of the lifespan of an ownership transfer.

Market of Resources Even though ownership relationships are represented as a hierarchy, for the duration of an ownership, the ownership is absolute. This means that owner of a resource decides independently whether to sell a resource during the interval in which it owns it, for a part of that interval. As a result, the interaction for trading in resources may assume a flat organization of owners and their owned objects. Owners may advertise objects that they own and for the intervals they own them for, and potential buyers may express interest in buying resources for intervals during which they are available. We do not restrict specific mechanisms for trading in objects in any way, except to assume that a market is established which enables buyers to buy and sellers to sell objects.

Hierarchy of Control Although terms of ownership of an object are decided between the seller and the buyer, distribution of identical objects between the owners is controlled by distribution controllers such as schedulers. How the objects are distributed is determined by composing layers of the hierarchy of terms of ownership under each domain of independent control - typically, a single machine.

At the same time, ownership is absolute for the duration of a certain time interval. In other words, a set of co-owned objects is encapsulated from other objects over a time-space interval.
3.4 Mobility of Computations

A computation may want to relocate when it does not (or does no longer) have access to something it needs, which is accessible only by relocating somewhere else. Relocation may be physical - from one control boundary to another - or economic - from one ownership boundary to another. Following are some examples of scenarios triggering relocation.

Better Resources. A computation may consider migration in pursuit of superior or more suitable resources for its execution. This may mean privileged databases, specialized protocols, secure network access and such, or better quality of service. Better quality of service may in turn be a function of the raw specifications of available resources, how well they are managed, probability of reliable access to them, and the cost it which they may be accessed.

Proximity to Data. Data required by a computation may not be accessible remotely at all or in a form that is meaningful to transfer. For example, the data may be in a raw unprocessed form which may incur exorbitant costs to transfer in terms of time or money. There may be data available only locally to computations admitted after strict admission control checks. There may be security or copyright concerns strictly limiting duration or type of access. Computations may not want the current host to know what it is accessing and from where. Data currency may be a concern, where it is best to be closest to the data; examples include operating system agents seeking a snapshot, or trader agents monitoring a stock market. Finally, an agent may need to collect empirical data about or available in a specific execution space. Any combination of above requirements may result in a need for a computation to migrate.

Application Requirements. Computations may have to relocate as a requirement of the application they are part of. This may be because some action is required at a distance, such as in a communications application or to rendezvous with computations located remotely. The location where the computation has to migrate may be absolute or relative to current state of the computation, including current locations of all computations. It may be important for a computation to keep migrating in order to avoid detection; this may, for instance, be important for an agent searching for intruders who are themselves trying to avoid detection. An agent may be scoping a network in search of something; for example, it may be scanning for signs of malicious activity, or it may be searching for other agents to pool resources with to solve a problem of common interest.
3.4.1 Patterns of Mobility

Enabling mobility of computations abstracted as mobile agents requires support for migration of computations, communications, their control structures, and for transfer of eCash. One or more of these actions may need to be carried out simultaneously. Listed below are some example patterns of mobility:

Remote Message A message has been created whose recipient is located in a different cyberorg. This would require the message to be transferred to the remote host and then the message to be delivered to the intended recipient.

Remote Processing of Message Agent should process an existing message after migrating to a different machine. This would require the agent and the message to be transferred to the remote host and then the message to be delivered to the agent.

Funded Service Request to Remote Agent Remote agent is to be requested to process a certain message. The remote agent expects financial reward for providing the service. Both the message as well the required amount of eCash need to be sent to the remote host.

Migration of Funded Agent Agent is to migrate to remote machine with funds to support itself. The agent and some eCash need to be transferred to the remote machine.

Extra Funds to Remote Agent An agent created locally and executing remotely has requested additional funds to complete its execution. It needs to be sent eCash.
Chapter 4

CyberOrgs

CyberOrgs is a model for resource sharing in a network of self-interested peers, where application agents may migrate in order to make avail of remotely located peer-owned resources. CyberOrgs organize computational resources as a market, and their control as a hierarchy. Specifically, each cyberorg encapsulates a concurrent computation (to be referred to as computations contained in the cyberorg), and resources available to it (to be referred to as resources owned by the cyberorg) for carrying out its computations or for resale. Cyberorgs may also be viewed as principals in a market of resources, where they can buy or sell resources among themselves using eCash in a shared currency.

Cyberorgs treat computational resources as being defined in time and space. Delivery of resources to cyberorgs is determined by a hierarchy of control decisions. In other words, cyberorg $a$ makes control decisions required for delivery of resources purchased from it by cyberorg $b$; cyberorg $b$ in turn makes control decisions determining how the resources purchased from it by cyberorg $c$ are to be delivered.

4.1 Control Hierarchy

A cyberorg represents a boundary around a computation and resources committed to its execution. Specifically, each cyberorg manages the execution of a multi-agent computation that needs to be completed using available resources. A cyberorg obtains resources by purchasing them from the cyberorg hosting it using its limited supply of funds, which we will call eCash. Cyberorgs are hierarchical, with resources flowing from the root to the leaves, and eCash flowing from cyberorgs
at the leaves toward the root. A cyberorg may not create eCash *ex nihilo*.

Cyberorgs may be seen as mobile traders and managers of computations and computational resources. Each cyberorg manages sets of (not necessarily independent) computations and (not necessarily physically co-located) resources, hosts other cyberorgs, and owns eCash with which it may purchase resources from other cyberorgs.

Cyberorgs organize a resource space as a tree of cyberorgs. Each cyberorg except the root cyberorg is contained inside another cyberorg. A cyberorg contained inside (i.e. hosted by) another cyberorg receives resources from its host in exchange for eCash payments, according to a pre-negotiated *contract*. A contract is negotiated between two cyberorgs when one of them wants to be hosted by the other. This contract stipulates the *types* and *quantities* of resources that will be available to the hosted cyberorg and their *costs*.

After satisfying its contractual obligations, a cyberorg distributes the remaining resources available to it among the computations it is managing according to a local *resource distribution strategy*.

### 4.2 Cyberorg Views

There are three different ways in which cyberorgs may be viewed:

- A *hierarchy* of cyberorgs with resources flowing from the root to the leaves and eCash flowing from the leaves to the root.

- A *set* of cyberorgs, each with “known” availability of resources, and known obligations to others.

- A *middleware* which stays invisible while facilitating the computations to proceed to completion.

We look at implications of each of these views in turn.

#### 4.2.1 Hierarchy of Cyberorgs

Each cyberorg receives its resources from its host cyberorg, creating a hierarchy. Figure 4.1 shows how computations — represented by black dots — and cyberorgs — represented by ellipses — may be
Figure 4.1: **Hierarchy of Cyberorgs**: Contracts stipulating availability of resources from cyberorg to cyberorg organize a system of cyberorgs as a tree. The delivery of resources is hence hierarchical.

seen as being contained inside other cyberorgs. This hierarchy is independent of the creator/created relationships between pairs of cyberorgs, or the relationships by which cyberorgs may sponsor other cyberorgs by sending them eCash. A cyberorg may host any number of cyberorgs inside it, so long as it has resources to satisfy its contractual obligations to them.

Intuitively, inclusion of one cyberorg inside another connotes a reliance for sustenance. Cyberorgs that do not have a direct or indirect containment relationship may have dependencies in terms of eCash: one cyberorg may have been created by another and may be “financially” supported by its creator. In contrast, a cyberorg contained in another cyberorg relies on a commitment by its host to sustain it according to the pre-negotiated contract. At the same time, the two cyberorgs are separated by cyberorg boundaries and have separate piles of eCash. The transfer of eCash outward in compensation for resources flowing in the opposite direction is determined by the contract.

These relationships have analogs in physical, biological and social systems in the real world. For example, consider relationship within a family with children of different ages. Older children capable of managing their pocket monies may be given an amount of eCash to manage themselves; younger children may be bought things by parents; a child yet to be born is physically sustained by the mother.
4.2.2 Set of Cyberorgs

Because a cyberorg enters the boundary of another cyberorg only after negotiating a contract with it, a system of cyberorgs may be seen as a set of cyberorgs with contractual relationships among them. Figure 4.2 shows this view of the cyberorgs shown in Figure 4.1. When a cyberorg arrives into a host, it does so with a promise of the resources that will be available to it and the cost for them. Because the contract stipulates availability and cost of resource, each cyberorg may be viewed as a microcosm with locally known resources and obligations.

This is significant in that it enables a cyberorg to assess the progress of computations it is managing with respect to resource availability, and consequently attempt to satisfy timeliness requirements. This is in contrast with conventional software systems where notions of timeliness in non-real-time applications are vague at best. In cyberorgs, resource availability as well as obligations to hosted cyberorgs are formalized as negotiated contracts.

Implications of cyberorgs’ ability to satisfy timeliness requirements on its resources are significant for the way timeliness is approached in software systems. Potential for timeliness is now a
function of not only the complexity of the problem being solved, but also of the amount of eCash held by a cyberorg. A “wealthy” enough cyberorg may now attempt to acquire the resources best suited for the timely completion of its computations.

eCash provides an objective basis for prioritization of computations. Where static prioritizing in traditional time-sharing systems separates computations over time through their life cycles, leaving total resource consumption often unbounded, eCash based prioritizing bounds the total amount of resources consumed by an application. Unlike static prioritizing, which give preference to privileged computations in time, using eCash enables separation and hence prioritization of computations in time as well as execution space. Most importantly, where traditional prioritizing cannot do better than committing all resources to a privileged computation, eCash based prioritizing can also migrate the computation to do better.

4.2.3 Transparent Middleware

Cyberorgs may also be viewed as a middleware in a facilitating role. As Figure 4.3 shows, cyberorgs exist as a separate layer of the underlying system, to provide an environment in which availability of resources is known and computations may proceed assuming this availability. In this way, cyberorgs achieve a separation of concerns between resource concerns of applications and actual functions of
the applications.

4.3 Trading in Resources

Computations may need amounts of resources as well as patterns in which resources are available – at an “affordable” cost – in order to complete successfully. Recall that for our purposes, successful completion requires timeliness. Contracts should thus be expressive enough to allow addressing such requirements.

4.3.1 Dynamic Pricing

Resource distribution schemes are often complicated by the need to adapt to demands for and availability of resources. We simplify this by separating the concerns for demand and availability from those of resource distribution. We achieve this separation by introducing a currency in which resources are sold and purchased. Specifically, even though each cyberorg has a set number of ticks at a time, and a fixed amount of eCash, the relationship between the two may change based on the demand and availability of resources. In other words, having a currency enables a dynamic pricing scheme.

4.4 Problem Classes

Computations that can benefit from employing the CyberOrg model broadly fall into three main categories. For each of these categories, certain conditions must hold for the CyberOrgs to be beneficial.

4.4.1 Remote Execution

CyberOrgs’ support for reifying control in a market of resources, enables computations to be carried out remotely. This appears to trivially improve upon the alternative of inability to compute remotely. However, the ability to compute remotely comes at the cost of control overhead required for supporting cyberorgs. Although for the computation requiring remote execution, there may be no alternative but to accept the overhead as the cost for flexibility, the existing computations on
the potential host may experience an unacceptable degradation in the quality of service. In other words, although cyberorgs enable better utilization of global resources, there is also a cumulative cost of the overhead.

From an individual computation’s perspective, the benefit of executing remotely must be balanced against the cost of executing remotely. Specifically, if the computation desires remote execution for quicker results, it must balance the benefit of migrating against the cost of doing so. Although the remote processor may be more powerful, the cost of migrating out and back may offset the savings in computation time.

4.4.2 Local Control

CyberOrgs enable systems to control resource consumptions of a computations executing on them. Here too it may be argued that the benefit is absolute in terms of the ability offer guaranteed resource availability to computations as opposed to the inability to do so. The overhead in this case, however, is of greater significance than the remote execution case. Specifically, if there are enough computations to achieve maximum utilization of the system, any overhead will take away from the total amount of computation that can take place in its absence. The need for control has to be carefully balanced against the need for resources.

4.4.3 Execution in Open System

The system’s ability to better control resource distribution results in predictable execution environments for individual computations. However, characteristics of a computation may preclude a benefit from such an environment. Although a computation may know what resources will be available to it, the question remains whether the same computation can know its resource requirements. In general, what percentage of computation benefits from such stringent control on resource consumption? What would this percentage have to be to make it viable to use CyberOrgs as the underlying execution environment?

There are specific types of computations that will obviously benefit from CyberOrgs, and those that will not. Approximation algorithms in which the quality of solution monotonically improves with the amount of resources invested, there is an inherent ability to stop at virtually any iteration
Insurance Company: An insurance company’s functions organized using cyberorgs. Agents created to handle newly received claims are encapsulated in cyberorgs with resources dedicated to handling the claims. In each cyberorg dedicated to a claim, new cyberorgs are created to handle specific aspects of the claim, which migrate to a remote cyberorg for research and return with results.

and report the result. However, that is not the case for a vast majority of computations where partial results are not useful. For such computations, it becomes important to either predict the actual resources that will be required for the computation to complete, or have a mechanism for re-negotiating the resources to be available. The former is not always possible. The latter can incur substantial overhead depending on a number of factors. For example, if the computation asks for only a small amount of additional resource, which repeatedly proves to be inadequate, there will be a large overhead. If, on the other hand, the computation asks for a large chunk of resource to avoid additional communication overhead, very little of it may end up being used, resulting in wasted resources.
4.5 Example: Insurance Company

Consider an insurance company handling claims (Figure 4.4). When a claim arrives, an agent is created for handling it. The company allocates resources for handling of the claim based on the complexity of the claim and the amount being claimed. In other words, the agent is placed in a new cyberorg with an amount of eCash which is a function of the complexity and amount of the claim. For simplification, let us assume that the complexity may be represented as a numeric value.

An insurance claim may have many aspects. If it is an automobile insurance claim, the costs involved may include medical and auto repair costs. There may also be a complex legal aspect of the claim. Handling each of these aspects may involve interaction with an external agency. The medical aspect may involve interaction with the hospital that provided medical treatment, the auto repair aspect may involve interaction with the auto repair shop. Each of these agencies is represented as a cyberorg with information resources which may be made available to clients requiring them.

The single agent inside a cyberorg handling a claim creates autonomous agents, each handling one aspect of the claim: medical, auto repair, legal. The cyberorg next isolates each of these agents into a new cyberorg, giving each an amount of eCash from its supply that is a function of the complexity of that aspect of the claim and the portion of the total claim that the particular aspect represents.

Cyberorgs hosting agents handling individual aspects negotiate contracts for the services they require with the service cyberorgs involved, and migrate to them. Service cyberorgs can commit only from their available resources. They commit total amounts of resources as well as their rates of availability. The cost they charge the clients for the service depends on the demand. While in a server cyberorg, the agents may communicate among themselves as necessary.

Once cyberorgs hosting aspects of a claim have finished researching those aspects at server cyberorgs, they migrate back to the claim’s cyberorg where they originated and they assimilate into the cyberorg, relinquishing autonomous control of their resources. After some final processing of the researched information, a decision is made on the claim.
Chapter 5

Operational Semantics

Our approach in formalizing cyberorgs is to separate concerns of computations from those of the resources required to complete them. Because our focus is on the usage of resources, we represent the resource requirements of each computation by the sequence of resources required to complete the computation. To simplify the model, we assume that resource requirements are known in advance. As an instantiation, we assume that the computations are carried out by systems of actors.

5.1 Actors

Actors are self-contained, interactive, autonomous components of a computing system that communicate by asynchronous message passing [2, 3]. The basic actor primitives are:

- \texttt{send}(a, v) creates a new message:
  - with receiver \( a \), and
  - contents \( v \)

- \texttt{newactor}(e) creates a new actor:
  - which is evaluating the expression \( e \), and
  - returns its address

- \texttt{ready}(b) captures local state change:
  - alters the behavior of the actor executing the \texttt{ready} expression to \( b \)
— frees that actor to accept another message.

These primitives form a simple but powerful set upon which to build further abstractions. Thus actors are a natural basis for a low-level language that supports a wide range of higher level abstractions and concurrent programming paradigms.

The actor newactor primitive extends the dynamic data creation capability in sequential programming languages by allowing creation of processes. The ready primitive gives actors a history-sensitive behavior necessary for shared data objects, by delineating a group of actions as atomic. This is in contrast to a purely functional programming model and generalizes the Lisp/Scheme/ML style sharing to concurrent computation. The send primitive is the asynchronous analog of function application. It is the basic communication primitive, causing a message to be put in an actor’s mailbox (message queue).

Using the three basic actor primitives, actor systems can be dynamically configured. New actors can be created and connections between actors can be made and broken as computation proceeds. Thus the model does not require that the structure or shape of a computational problem be completely determined, or that the execution resources be fixed, before work on solving it can be initiated.

Actors provide a natural extension of the object-oriented paradigm to concurrent and distributed computation. They support encapsulation, description as behavior templates, and re-usability via libraries accessed using message-passing protocols. The locality properties of actors guarantee that changes of representation and elaborations can be made independent of the interaction with, and behavior of, other actors. Thus actors can support local instrumentation and monitoring which provide important tools for analysis and debugging.

5.1.1 Actor Syntax and Semantics

It is possible to extend any sequential language with actor constructs. For example, the call-by-value λ-calculus is extended in [5].

Instantaneous snapshots of actor systems are called configurations; an actor computation is defined as a labeled transition relation on configurations. The notion of open systems is captured by defining a dynamic interface to a configuration, i.e., by explicitly representing a set of receptionists
which may receive messages from actors outside a configuration and a set of actors external to a configuration which may receive messages from the actors within.

**Definition (Actor Configurations):** An actor configuration with actor map, $\alpha$, multi-set of messages, $\mu$, receptionists, $\rho$, and external actors, $\chi$, is written as

$$\langle \alpha \mid \mu \rangle^\rho_\chi$$

where $\rho, \chi$ are finite sets of actor addresses, of local actors known outside the configuration and external actors known within the configuration, respectively, $\alpha$ maps a finite set of actor addresses to their behaviors, $\mu$ is a finite multi-set of (pending) messages. Let $A = \text{Dom}(\alpha)$, i.e., the domain of $\alpha$, then the following must hold:

(0) $\rho \subseteq A$ and $A \cap \chi = \emptyset$,

(1) if $a \in A$, then $\text{FV}(\alpha(a)) \subseteq A \cup \chi$, where $\text{FV}(\alpha(a))$ represents the free variables of $\alpha(a)$; and if $\langle v_0 \leftarrow v_1 \rangle$ is a message with content $v_1$ to actor address $v_0$, then $\text{FV}(v_i) \subseteq A \cup \chi$ for $i < 2$.

For an actor with address $a$, we indicate its state as $[e]_\alpha$, when it is busy executing $e$. The
identifier $e$ represents the actor’s current (local) processing state.

We can extend the local transitions defined for a sequential language ($\to^\lambda$), by providing labeled transitions for the actor program as follows (assume that $R$ is the reduction context in which the expression currently being evaluated occurs). For brevity, we skip writing the labels corresponding to each transition unless needed.

**Definition ($\to^\lambda$):**

$$e \xrightarrow{\text{Dom}(a) \cup \{a\}} e' \Rightarrow \langle \alpha, [e]_a \mid \mu \rangle_\chi \xrightarrow{\lambda} \langle \alpha, [e']_a \mid \mu \rangle_\chi$$

If expression $e$ reduces to expression $e'$ in the context of $\text{Dom}(a) \cup \alpha$, then if $e$ is the behavior of actor $a$, the behavior will change to $e'$ at the end of the transition.

$$\langle \alpha, [R[\text{newactor}(e)]]_a \mid \mu \rangle_\chi \xrightarrow{\lambda} \langle \alpha, [R[a']][a] \mid \mu \rangle_\chi \quad a' \text{ fresh}$$

$$\langle \alpha, [R[\text{ready}(v)]]_a \mid \mu, <a \Leftarrow cv> \rangle_\chi \xrightarrow{\lambda} \langle \alpha, [\text{app}(v, cv)]_a \mid \mu \rangle_\chi$$

$$\langle \alpha, [R[\text{send}(u_0, v_1)]]_a \mid \mu \rangle_\chi \xrightarrow{\lambda} \langle \alpha, [R[\text{nil}]]_a \mid \mu, <u_0 \Leftarrow v_1> \rangle_\chi$$

Transitions for newactor, ready, and send carry out the corresponding primitive operations. newactor($e$) results in creation of an actor with a new name and with the behavior $e$. The context in which the command is invoked receives the name of the new actor. ready($v$) in the actor’s execution context sets up the actor to apply behavior $v$ to the next message received. When there is a communicable value $cv$ in a message to the actor, the message is received and the behavior is applied to the $cv$. send($u_0, v_1$) simply results in creation of the message $v_1$ to the actor $u_0$. 

34
\[\langle \alpha \mid \mu, m \rangle_\chi^\rho \mapsto \langle \alpha \mid \mu \rangle_\chi^{\rho'}\]

if \(m = \langle a \leftarrow v \rangle, a \in \chi\), and \(\rho' = \rho \cup (FV(v) \cap \text{Dom}(\alpha))\)

\[\langle \alpha \mid \mu \rangle_\chi^\rho \mapsto \langle \alpha \mid \mu, m \rangle_\chi^{\rho \cup (FV(v) \cap \text{Dom}(\alpha))}\]

if \(m = \langle a \leftarrow v \rangle, a \in \rho\) and \(FV(v) \cap \text{Dom}(\alpha) \subseteq \rho\)

The final two transitions represent communication into or out of the configuration. The first transition represents the case when there is a message directed to an external actor. In this case, if there are names of the configuration’s actors contained in the message being sent, the corresponding actors become receptionists. The second transition represents the case when a messages is received by a receptionist from an external actor. In this case, any names of actors contained in the message which are not local actors, are added to the set of external actors.

5.2 Instantiating CyberOrgs with Actors

An instantaneous snapshot of a system of cyberorgs is represented by \(\langle \Gamma || \mathcal{M} || \mathcal{C} || \Theta || \mathcal{D} \rangle\). \(\Gamma\) represents the set of cyberorgs in the system, \(\mathcal{M}\) is a name table which maps cyberorgs and actors to cyberorgs hosting them, \(\mathcal{C}\) is the set of contracts between client cyberorgs and their host cyberorgs, \(\Theta\) is the multiset of directed resources in the system, and finally \(\mathcal{D}\) is a matrix which keeps track of distances between cyberorgs and actors.

The hosting relationship between two cyberorgs is not represented by syntactic containment. Instead, \(\mathcal{M}\) maintains information about which cyberorg each cyberorg is hosted by. However, the containment of actors within cyberorgs is represented as syntactic containment as shown later.

We abstract computational resources as ticks, which determine the granularity of availability and consumption of resources. In other words, resources are provided to cyberorgs in terms of numbers of ticks, and computations consume resources in multiples of ticks. Ticks are defined in time and space and are sequentially ordered. If a tick is available in a cyberorg which cannot consume it, or if it is available at a time at which it cannot be consumed, it expires. Because a tick is the basic unit of resource, introduction of ticks into the system defines the system clock.
The clock advances as ticks are introduced; consequently, absolute rates of availability of ticks to cyberorgs are with respect to the introduction of ticks into the system.

Because the model abstracts over physical machines, distances among cyberorgs, among actors, and between cyberorgs and actors are explicitly represented with $D$. The distance from a cyberorg is defined only when all actors of the cyberorg are at zero distance from each other. In other words, if a cyberorg is distributed over a distance, it's distance from actors or other cyberorgs is undefined.

Each cyberorg in the system is represented by $\mathcal{C}_a, \mu, \$^\xi_{\mathcal{C}_a} \omega$, where $\alpha$ represents a set of actors whose computation is managed by the cyberorg; $\mu$ is a set of actor messages with local or remote recipients; $\$^\xi$ is the amount of eCash in the cyberorg; $\xi$ and $\omega$ are the resources required by the cyberorg for execution, and those offered by it to potential clients, respectively; and $c_i$ is the cyberorg's unique name.

The current state of an actor in the model, at any instant, is represented by a state in the future and the resources required to reach that state. Specifically, actor $a$’s state is written as $[n \circ s]_a$, where $s$ is the state it would reach after receiving $n$ ticks. An actor is said to have reached a state when the count of ticks required has reached zero.

5.2.1 Progress

Progress in a system of cyberorgs is represented by transitions occurring with introduction of ticks into the system. When a tick is inserted into a cyberorg, the cyberorg may pass it on to a client cyberorg, it may use it for progressing on one of its actors. Whether a tick is passed on to a client or used locally depends on the contracts that the cyberorg has with its clients.

Contracts determine the total number of ticks that the clients must receive, and the rates at which they must receive them. Rates of receipt of ticks are as a ratio of the total number of ticks inserted into the system at the root. Contracts also determine the costs which the clients are supposed to be charged for the ticks.

Surplus ticks after a cyberorg’s contractual obligations to its clients have been satisfied, may be distributed among the local actors.

When the function $T_{c_1}$ representing the decision process for cyberorg $c_1$ returns actor $a$ as the recipient of the tick $t(c_1)$ for $c_1$, given the cyberorg’s state $st(c_1)$ and its set of contracts $co(c_1)$, the
system progresses by decrementing the number of ticks to a’s next state from $n$ to $n - 1$.

$$\langle\langle [n \circ e]_\alpha, \alpha, \mu, S\epsilon_{\alpha}, \Gamma, M[C]\rangle, t(c_1), \Theta \mid \Delta \rangle$$

$$\rightarrow \langle\langle [n \circ (n - 1) \circ e]_\alpha, \alpha', \mu', S\epsilon'_{\alpha'}, \Gamma, M[C] \mid \Theta \mid \Delta \rangle$$

$$T_\alpha(st(c_1), co(c_1)) = a, \ n > 0$$

As a result of delivery of this tick to actor $a$, new actors and messages may be created in the cyberorg, changing the set of other actors from $\alpha$ to $\alpha'$, and the multiset of messages from $\mu$ to $\mu'$. Similarly, the resource offerings and requirements of the cyberorg may also change.

When an actor’s number of ticks required to reach the next state reduces to zero, the state is said to be reached. At this point, the state may be rewritten to reflect the number of ticks required to reach the following state $\epsilon'$. Because this is a rewriting of the current state rather than a change of state, the transition does not require any ticks to carry out.

$$\langle\langle [0 \circ e]_\alpha, \alpha, \mu, S\epsilon_{\alpha}, \Gamma, M[C] \mid \Theta \mid \Delta \rangle$$

$$\rightarrow \langle\langle [n \circ e']_\alpha, \alpha', \mu', S\epsilon'_{\alpha'}, \Gamma, M[C] \mid \Theta \mid \Delta \rangle$$

If there is a cyberorg $c_2$ being hosted by $c_1$ and the new tick is to be passed on to $c_2$, then, the tick is redirected to $c_2$, and an amount of eCash $\Delta$ representing the cost of the tick – determined by the contract $co(c_1, c_2)$ between $c_1$ and $c_2$, and $st(c_1)$, the state of $c_1$ – is transferred from $c_2$ to $c_1$.

$$\langle\langle c_1, \mu_1, S_1\epsilon_{c_1}, [c_2, \mu_2, S_2]_{\epsilon_{c_2}}, \Gamma, M[C] \mid \Theta \mid \Delta \rangle$$

$$\rightarrow \langle\langle c_1, \mu_1, S_1 + \Delta \epsilon_{c_1}', [c_2, \mu_2, S_2 - \Delta \epsilon_{c_2}], \Gamma, M[C] \mid t(c_2), \Theta \mid \Delta \rangle$$

if $< c_1, c_2 > \in M, T_\alpha(st(c_1), co(c_1)) = c_2$

where $\Delta = cost(st(c_1), co(c_1), c_2)$

If there are no active actors or cyberorgs to be given a tick, the tick expires:

$$\langle\langle \alpha, \mu, S\epsilon_{\alpha}, \Gamma, M[C] \mid t(c_1), \Theta \mid \Delta \rangle$$

$$\rightarrow \langle\langle \alpha, \mu, S\epsilon'_{\alpha}, \Gamma, M[C] \mid \Theta \mid \Delta \rangle$$

$$T_\alpha(st(c_1), co(c_1)) = \phi$$
5.2.2 CyberOrg Primitives

In addition to transitions corresponding to progress in actor computations, there are a number of transitions in the system which correspond to CyberOrg primitives. These transitions happen through invocation of CyberOrg commands from helper actors, which in turn are created by the cyberorg’s facilitator actor. A facilitator actor monitors the state of the current host as well as the cyberorg’s resource requirements, and creates helpers to carry out CyberOrg primitives. Facilitators and helpers are different from application actors hosted by a cyberorg in that they do not have names, and hence, may not receive messages from other actors. They also do not participate in the computations pursued by the application actors. Finally, because no actors have helpers’ names, they safely disappear from the system after carrying out their operations.

Creation and Absorption

As illustrated in Figure 5.2, a new cyberorg is created by using the isolate primitive, which collects a set of actors, messages, and electronic cash, and creates a new cyberorg hosted locally. The construct isolate takes as parameters a subset of the actors in $c_1$, $\alpha_2$, a subset of the messages, $\mu_2$, and a part of its cash $\$_2$, as well as representations of what the new cyberorg ought to offer other cyberorgs and require from other cyberorgs (currently $c_1$ itself), and creates a new cyberorg inside $c_1$’s boundaries with a fresh name $c_2$. As a result, a new entry is placed in the name table depicting $c_2$’s presence inside $c_1$, and entries for locations for actors inside $c_2$ are modified to depict the change.

\[
\langle \{\langle \{\text{isolate}(\alpha_2, \mu_2, \$_2, \chi_2, \xi_2, \omega_2)\rangle_{\rho}, \alpha_1\}, \mu_1, \$_1\} \mid \chi_1^{\omega_1}, \Gamma \mid \mathcal{M} \mid \mathcal{C} \mid \Theta \mid D \rangle \\
\rightarrow \langle \{\alpha_1 - \alpha_2, \mu_1 - \mu_2, \$_1 - \$_2\} \mid \chi_1^{\omega_1}, \{\alpha_2, \mu_2, \$_2\} \mid \mathcal{M} \mid \mathcal{C} \mid \Theta \mid D \rangle \\
\alpha_2 \subset \alpha_1, \mu_2 \subset \mu_1, \$_2 \leq \$_1, \text{\$}_2 \text{ fresh}
\]

As shown in Figure 5.3, a cyberorg disappears by assimilating into its host cyberorg using the asault primitive, relinquishing control of its contents - actors, messages and eCash - to its host. The assimilating cyberorg disappears, and its host becomes the container for its contents.
Figure 5.2: **Cyberorg Isolation:** A cyberorg encapsulates a number of its actors, messages and an amount of its eCash as a new cyberorg with independent control. The isolating cyberorg becomes the host of the new cyberorg and has a contract by which the new cyberorg will be served.

Figure 5.3: **Cyberorg Assimilation:** A cyberorg assimilates into its host cyberorg by relinquishing independent control of its eCash, actors and messages. The contents of the assimilating cyberorg become contents of the cyberorg originally hosting it.
Assimilation of a client cyberorg into its host can potentially be a dangerous operation to allow. Although the primitive hands the client’s eCash to the host to use at its discretion, its computations also join the host’s computations and may interact or interfere in undesirable ways. The host is however protected because it alone decides whether the assimilated cyberorg’s computations are allowed to advance in their processing. In other words, when a cyberorg decides to assimilate into its host, it relinquishes all control over its contents: its contract with the host dissolves, its eCash is added to the host’s eCash, and its computations may or may not receive any ticks from the host without any contractual obligations.

Mobility

A facilitator may realize that its resource requirements exceed what is available by its contract with the host cyberorg. As a result, it creates a helper to search for alternate hosts:

Cyberorgs 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:\(^1\)

1. Search for a potential host. This makes use of the yellow page services provided by the system to search for cyberorgs which may offer needed ticks for an acceptable price.

\(^1\)Migration of a part of a cyberorg’s computation would require isolation first.
2. **Negotiate** a contract with potential hosts. Negotiation involves interaction with potential hosts for possible access to their ticks. Negotiation may be initiated by a cyberorg wanting to migrate itself or wanting to migrate part of its computation. On successful culmination of a negotiation, a contract is reached with a potential host cyberorg, which would hold between the migrating cyberorg and the host.

\[
\langle\{0 \circ \text{negotiate}(C)\}_{\phi, \alpha_1}, \mu_1, \$1\rangle_{\mathcal{C}_1}^{\mathbb{Q}_1}, \langle\mu_2, \$2\rangle_{\mathcal{C}_2}^{\mathbb{Q}_2}, 1|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D} \rangle
\]

\[
\rightarrow \langle\{\mathcal{C} \circ \text{migrate}(c_2)\}_{\phi, \alpha_1}, \mu_1, \$1\rangle_{\mathcal{C}_1}^{\mathbb{Q}_1}, \langle\mu_2, \$2\rangle_{\mathcal{C}_2}^{\mathbb{Q}_2}, 1|\mathcal{M}|\mathcal{C}'|\Theta|\mathcal{D} \rangle
\]

where \(C \subseteq C = \{c_k, c_{k+1}, \ldots, c_l\}\) such that \(x < \omega_2, C' \supseteq C\)

If there are no cyberorgs which can serve \(c_1\)'s resource requirements, no negotiation can happen, and \(c_1\) adapts to its current resource availability:

\[
\langle\{0 \circ \text{negotiate}(C)\}_{\phi, \alpha_1}, \mu_1, \$1\rangle_{\mathcal{C}_1}^{\mathbb{Q}_1}, 1|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D} \rangle
\]

\[
\rightarrow \langle\alpha_1, \mu_1, \$1\rangle_{\mathcal{C}_1}^{\mathbb{Q}_1}, 1|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D} \rangle
\]

if \(C = \phi \land \forall c_i \in C = \{c_k, c_{k+1}, \ldots, c_l\}\) such that \(x < \omega_i\)

3. **Migrate** to the selected host. If a contract has been successfully negotiated, a client can relocate to the host using the migrate primitive as shown in Figure 5.4.

\[
\langle\{0 \circ \text{migrate}(c_2)\}_{\phi, \alpha_1}, \mu_1, \$1\rangle_{\mathcal{C}_1}^{\mathbb{Q}_1}, \langle\mu_2, \$2\rangle_{\mathcal{C}_2}^{\mathbb{Q}_2}, 1|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D} \rangle
\]

\[
\rightarrow \langle\alpha_1, \mu_1, \$1\rangle_{\mathcal{C}_1}^{\mathbb{Q}_1}, \langle\mu_2, \$2\rangle_{\mathcal{C}_2}^{\mathbb{Q}_2}, 1|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D} \rangle
\]

\[
\text{if}\{\text{co}(c_1, c_2)\} \in C
\]

where \(\mathcal{M}'\) reflects the change in location of \(c_1\)

---

\(^2\)A migrating cyberorg may not exist at the time of negotiation; it may be created following a successful negotiation.
Figure 5.4: Cyberorg Migration: A cyberorg migrates from its current host cyberorg to another cyberorg. This happens after a prospective host has been found and a contract has already been negotiated with it.

where $D'$ is the revised distance matrix representing any changes in distances that might have occurred as a result of migration.

If a contract was not successfully negotiated, and $c_1$ adapts to its current resource availability:

\[
\langle \left\{ 0 \right\} \circ \text{migrate}(c_2), \alpha_1, \mu_1, s_1 \rangle_{c_1}^{\mu_2}, \langle \alpha_2, \mu_2, s_2 \rangle_{c_2}^{\mu_2}, \Gamma | \mathcal{M} | \mathcal{C} | \Theta | D \\
\rightarrow \langle \alpha_1, \mu_1, s_1 \rangle_{c_1}^{\mu_2}, \langle \alpha_2, \mu_2, s_2 \rangle_{c_2}^{\mu_2}, \Gamma | \mathcal{M} | \mathcal{C} | \Theta | D \\
\text{if} \{ \text{co}(c_1, c_2) \} \notin \mathcal{C}
\]

The process of search, negotiation and migration is illustrated in Figure 5.5

5.3 Design Decisions

In this section, we discuss the various design choices made in designing the CyberOrg model.

5.3.1 Representation of Actors

We did not want to model actor computations in this model, only their resource needs. An earlier attempt to extend the semantics of actors with actor-level migration and resource exchange prim-
Figure 5.5: Remote Message: A message created by an actor in a cyberorg directed at an actor located inside another cyberorg is encapsulated into a new cyberorg. The new cyberorg migrates to the remote cyberorg containing the intended recipient actor, and assimilates into the host cyberorg on arrival, breaking its encapsulation and turning the message into a local messages which can be delivered.
itives [4] had increased the model's complexity significantly. Having decided to represent resource requirements of actors rather than their computations, we had two choices. One choice was to estimate resource needs of actors, and if they prove to be wrong, have mechanisms for returning excess resources or provide extra resources to enable completion (as in [45]). The other choice was to assume that a cyberorg could accurately tell the resource requirements of its actors, precluding later corrections. We made the second choice for two reasons. First, though operationally different, guessing and then correcting essentially approximates guessing correctly the first time, though with an added delay. Because the delay caused by the correction cannot be known, we may as well ignore it to simplify the model. Second, it does accurately model a policy where actors are awarded as many ticks as their computations are worth to the cyberorg; they are supposed to be starved after they've used up these resources.

Additionally, actor creations, and message creations are modeled as consequences of an actor advancing as a result of consuming a tick. Consumption of a tick results in an actor's progress, at the end of which, new actors and messages supposed to have been created during that progress are deemed created. Tick consumptions are atomic transitions; so creations of actors and messages as a consequence are modeled to have happen by the end of the transition.

5.3.2 One Tick at a Time

Cyberorgs advance as a result of insertion of ticks into the system, one tick at a time. On receiving a tick, a cyberorg determines whether to pass it on to a client cyberorg, one of its computations, or use it to perform one of its system tasks, based on the contracts it is obliged to honor, and needs of its local computations it must complete.

Insertion of ticks one at a time, and their expiration when no computation is ready to use them is a simpler representation of the way processor resource becomes available, than granting of an approximate number of ticks followed by corrective mechanisms. It may also be viewed as a more accurate modeling in an ideal world where resource needs are known a priori or may be dictated.

From a modeling standpoint, we argue that the insertion of ticks one at a time, and their expiration when no computation is ready to use them, is a simpler representation of the way a variety of computational resources become available. It may also be viewed as a more accurate
model in an ideal world where resource needs are known \emph{a priori} or in applications where they may be dictated, as in the case of Design-to-Criteria systems \cite{64}. Furthermore, modeling resources as expiring ticks which become available one at a time, allows using the rate at which resources become available to the system as a basis for managing rates of availability of resources to cyberorgs. In other words, cyberorgs may offer absolute rates of availability of resources to potential clients as functions of ticks becoming available to the root cyberorg.

It is acknowledged that for an implementation to manage processor ticks one at a time, would mean incurring prohibitive overheads. Transfers of ticks from server cyberorgs to client cyberorgs - which are stipulated by contracts - may be easily optimized because known contracts rather than needs of clients determine how many ticks to provide. Awarding of ticks by cyberorgs to the computations they manage may be approximated by using a scheme that guesses and then adjusts as necessary.

5.3.3 CyberOrg Structure

In their first incarnation, cyberorgs were mobile actors with eCash, which could migrate from one resource ownership domain (ROD) to another. These RODs were represented by resource managers with which the cyberorgs would negotiate their terms of hosting with. The more general multi-threaded structure of cyberorgs came out of a realization that there was no reason why boundaries of concurrency should be identical to the boundaries of resource consumption; cyberorgs could determine resource consumption boundaries around general computations.

Separation of cyberorg boundaries from actor boundaries initially had the effect of adding complexity to the model. However, having multiple threads executing inside a cyberorg led to the observation that after being committed certain resources, a cyberorg may end up with surplus resources which it may want to sell. A generalization of this idea led to the disappearance of RODs from the model. RODs were replaced with cyberorgs, and cyberorgs became a hierarchy.

Disappearance of RODs as entities distinct from cyberorgs simplified the model significantly. Instead of separate sets of interactions between cyberorgs, and between cyberorgs and RODs, now there were only one type of interactions: between cyberorgs.
5.3.4 CyberOrg Primitives

CyberOrg primitives evolved along with the evolution of the structure of cyberorgs: from the complex primitives for interactions between cyberorgs and RODs to a much simpler set.

There were two types of interesting resource-oriented primitives identified in [4]:

- Variations of `create` and `send` primitives which allowed specification of eCash to be given to the newly created cyberorg, or to be sent to the recipient of a message.

- Primitive for migration of a cyberorg from one ROD to another.

It was soon realized that there were a variety of interesting operations which although can be implemented using this set of primitives, it would be cumbersome and somewhat unnatural to do so.

With the generalization of CyberOrgs to their present simpler form, it became possible to identify a more general purpose of defining CyberOrg primitives: to transfer an autonomous funded computation from one environment to another. It was possible to achieve this in multiple ways. We considered the option of using three primitives: `split`, `merge`, and `migrate`. A cyberorg could split into two, it could merge with another cyberorg, or it could migrate to another cyberorg. Sequences of these primitives would be used to achieve desired effects. For example, if a message is generated inside one cyberorg which is intended for an actor located in another cyberorg, the sender’s cyberorg will split into two, one containing only the remote message, and the other containing remaining contents of the splitting cyberorg. The message cyberorg could then migrate to the host of the recipient’s cyberorg, where it would merge with the cyberorg to make the message local.

The problem was determining when two cyberorgs are supposed to merge. Although the solution was straightforward for a merge accomplishing any one type of task (such as delivering a remote message), coming up with a general scheme demanded too much from the underlying system. Specifically, in the most general case, the underlying system would have to identify all potential candidates for merger with each cyberorg.

The other alternative was to consider another set of three primitives: `isolate`, `assimilate`, and `migrate`. These three primitives enabled mobility of a computation from one cyberorg to another, without the high demands on the system. Here the complexity of choosing whom to merge with -
which had to be enabled by the system - in the previous alternative, transforms into the choice of where to migrate. This choice can be supported by the system through a yellow pages service in which cyberorgs may advertise their offerings.

5.3.5 Expiration of Ticks

If a tick is received by a cyberorg when it does not have a computation or a cyberorg that can use it, the tick expires. This is consistent with the nature of many resources. For example, processor cycles can be used only if there is a task that can use them; they cannot be saved for future use. If there are no tasks ready for execution, the cycles pass unutilized. Similarly for network bandwidth. If communications are not ready to proceed at a time when there is idle bandwidth, it goes to waste; it may not be saved for future use.

5.3.6 Ownership

CyberOrgs specifically separate resource control from ownership, distinguishing the model from the closely related Quantum model [45, 46]. An important feature of CyberOrgs is that it offers a basis for exchange of resources between autonomously managed and funded computations, rather than management of resources within a single-owner domain - a function subsumed by CyberOrgs. The CyberOrgs model differentiates between measures of resources and the currency through which they are exchanged. This specifically allows the use of dynamic pricing schemes naturally. Unlike leaf groups in Quantum, a leaf cyberorg does not necessarily entirely reside on a single physical node. Both leaf and non-leaf cyberorgs may migrate to other cyberorgs on the same host (or set of hosts) or different. Relationship between a client cyberorg and its host cyberorg is determined by a service contract negotiated between the two prior to the client’s arrival, unlike in Quantum where the terms of relationship are predetermined or decided by the parent group. Instead of awarding quantities of energy (requiring mechanisms for absorption of unused energy and provision of more energy at exhaustion), cyberorgs progress as a result of insertion of ticks into the system. On receiving a tick, a cyberorg determines whether to pass it on to a client cyberorg, to one of its computations, or to use it to perform one of its system tasks. This determination is based on the contracts a cyberorg is obliged to honor, the needs of its local computations that it must complete.
5.4 Properties

Proposition 5.1 The amount of eCash in the system remains constant.

Proof: Money is only transferred in the system; it is never consumed.

Proposition 5.2 If no new ticks are inserted into the root cyberorg, the system will eventually reach a dormant state.

Proof: Proof is by induction.
Base: A tick intended for a leaf cyberorg:

- is either forwarded to an actor which consumes it and advances one step and stops
- or expires because the cyberorg cannot use it.

The tick is either consumed or expires, preventing the system from further progress.
Inductive step: Take a non-leaf cyberorg such that the proposition is true for all its client cyberorgs. A tick intended for this cyberorg:

- is either forwarded to an actor which consumes it and advances one step and stops
- or forwarded to a client cyberorg, in which it is consumed, and the cyberorg stops.
- or expires because the cyberorg cannot use it.

Proposition 5.3 If transfer of eCash is limited to purchase of ticks, cyberorgs carrying eCash only migrate up the cyberorg tree, and the price of ticks only reduces in fixed quantaums, then the system will eventually reach a dormant state.

Proof: The system progresses as a result of transfer of ticks down the tree, in exchange for eCash moving up the tree. If all non-purchase movement of eCash is up the cyberorg tree, eventually, either all computations would have either completed, or run out of eCash.

Proof is by induction.

\textsuperscript{3}Similar in principle to impossibility of Zeno machines: one cannot do infinite amount of computation with finite amount of eCash
Base: The amount of eCash held by a leaf cyberorg will monotonically decrease so long as it has active actors or it transfers eCash to other cyberorgs above it. If all actors are inactive, the cyberorg will wait (possibly indefinitely), until one of them becomes active. If they never become active, the cyberorg has become dormant. If they become active, eCash will be consumed. Once all eCash has been consumed, the cyberorg becomes dormant because it has no clients to give it eCash, and consequently, there is no way for it to purchase more ticks.

Inductive Step: Consider a non-leaf cyberorg such that the proposition is true for all its client cyberorgs. This cyberorg’s eCash can increase as a result of transfer of eCash from clients. Its eCash can decrease if it has active actors or if it transfers eCash to cyberorgs higher up the hierarchy. In either case, so long as there are active actors in the subtree rooted at this cyberorg, the amount of eCash contained in the subtree is monotonically decreasing. Either it will run out of active actors or eCash, making the cyberorg dormant in either case.

5.5 Discussion

CyberOrgs may be extended in a number of ways to increase the model’s expressive power. Here are two specific examples of such extensions.

5.5.1 Typed Ticks

Although ticks are a generic resource in the model, they may be typed; that is, different types of resources may be represented in the system. For example, migration over networks would require network ticks with the appropriate attributes. The types of ticks would depend on the resources that need to be modeled and the granularity at which they are modeled. For example, ticks for representing processing power may denote units of processor time, or the number of operations. Depending on a tick’s type, the type may also have attributes. A network tick representing a unit of network resources would typically have attributes denoting the source and the destination of the network link.
5.5.2 Resource Consumption Patterns

It is safest for cyberorgs not to commit ticks which they have already committed elsewhere in contracts; however, it can also be wasteful. Given the difficulty of guessing the resources and the rates of availability at which a client would require them, typically there are committed resources — especially rate of availability commitments — which go unused.

An extension of cyberorgs can use probabilistic estimates of actual resource use by clients to decide when it is safe to over-commit. This could be done by allowing probabilities into contracts. Additionally, the model could support mechanisms for imposing penalties on a server for negligently over-committing its ticks.
Chapter 6

Implementation

A prototype cyberorg implementation has been built using Actor Foundry [36], a library of Java classes supporting Actor functionality. Actor Foundry is meant to be a research tool, and hence is designed with the goals of modularity and extensibility rather than pure efficiency. Actor programs may be written and executed in a run time that supports operational semantics of the Actor model. Specifically, cyberorgs are implemented as a library of Actor Foundry code.

6.1 System Design

We look at the artifacts that make up a system of cyberorgs, the control mechanisms which carry out primitive CyberOrg operations, strategies that a programmer may implement, and the user interface for launching computations and interacting with them.

6.1.1 Artifacts

A cyberorg system consists of actors carrying out application tasks (which we will call application actors), and cyberorgs managing resource utilization of the actors. The unit of resource is a tick. Each method of an actor requires a number of ticks to execute, depending on the parameters passed to it. Ticks are transferred between cyberorgs based on contracts between them. These contracts are negotiated bilaterally between the cyberorgs. Ticks received by a cyberorg are autonomously distributed by the cyberorg among the application actors it manages. Ticks received by an actor may be used by the actor for invoking any of the methods ready for invocation (as a result of received messages). Both cyberorgs and actors use their own tick distribution/utilization strategies.
To implement an application as a system of cyberorgs, a programmer writes classes defining the cyberorgs, and the application actors. A cyberorg class definition identifies events that would trigger cyberorg behaviors, as well as defines these behaviors in terms of basic cyberorg primitives. Application actor classes are similar to actor classes in *Actor Foundry*, except that for each method defined, the class contains another method to compute the resource requirements of executing the method with the passed parameters.

Cyberorgs may pick from a number of available negotiation protocols to use for negotiating contracts. The two negotiating parties would communicate using a basic pre-negotiation protocol to decide which negotiation protocol to use. These protocols may either be provided as part of a library, or they may be written by a programmer especially for a particular application.

### 6.1.2 Control

Sequences of cyberorg primitives may be triggered for application or resource related reasons. In this simulation, we consider three specific tasks for which these primitives are triggered: isolation, assimilation, and migration.

#### Isolation

A cyberorg may decide that it would like to create independent control for a set of its clients and application actors. This may be reasonable to do for a variety of reasons. For example, the complexity of deciding how to distribute resources among a large group of actors may be too high. In such a case, it would make sense to identify actors that are closely working together and to create a cyberorg to manage them.

Similarly, it may be that alternative solution paths to a problem are being pursued, and each subtree is to be given an equal amount of resource. Consider, for instance, a resource bounded version of depth first search. In this case as well, it would make sense to create separate cyberorgs for each of the subtrees, each with an amount of eCash to use as it pleases.
Assimilation

Assimilation of a cyberorg into its host may be seen as an act of relinquishing one’s autonomy. Assimilation may happen as a result of resource-related reasons or for application-related reasons.

An example of an application-related reason for assimilation is assimilation of a cyberorg containing an actor message intended for a local actor. Recall that such a cyberorg would be created at another cyberorg, one of whose application actors would have generated a remote message.

A cyberorg may also assimilate because it finds it unnecessary to maintain independence, which entails the need to manage application actors and resources. If, for example, a cyberorg has only a handful of actors and no client cyberorgs, the overhead of independent management of these resources may be too exorbitant. However, giving up independence amounts to giving it up completely. For instance, the host cyberorg may choose to take all the eCash and never schedule the actors assimilating into its environment\(^1\). This may, in fact, be an effective mechanism for a cyberorg to cause a garbage collection of its actors. But recall that for a garbage collection of actors to conform with Actor semantics, it should not be possible for any messages to be targeted at such actors.

Migration

Migration too may happen for application reasons or for reasons of resource availability. Here we present some examples of cyberorg migration.

Remote Messages The most common application reason for migration is to relocate a message intended for a remote actor to its recipient’s cyberorg. The process for this type of migration is triggered by generation of a remote message by an actor. On being identified as a remote message, the cyberorg creates a new cyberorg enveloping this message and an amount of eCash necessary for processing the message, and sets it up for migration to the remote cyberorg.

Locality Some actors may need to have extensive interaction with other actors located remotely. These actors may be isolated into a new cyberorg and programmed to migrate to the cyberorg containing the remote actors, and assimilate.

---

\(^1\)From a host cyberorg’s perspective, the choice to do this amounts to being able to protect itself from malicious or erroneous computations entering it.
Note, however, that locality in terms of cyberorgs is not identical to locality in terms of machines. Communication between actors contained in different cyberorgs located on the same machine may often not incur enough additional cost to warrant migration.

**Resource Conditions** Migration may also be triggered by changes in resource conditions. Resource conditions are measured with respect to the cyberorg's requirements, which in turn depend on its contractual commitments, requirements of its own application actors, and affordability of resources.

Because changes in resource conditions are relative to cyberorg needs, they can come about as a result of diminished resource availability or an increase in resource requirements. Unless cyberorgs commit resources to clients based on probabilistic measures, it is not possible for contractual obligations to result in increased requirements. Application actors, however, may make higher resource demands through the numbers of messages they generate, the number of new actors they spawn, or simply as a result of a computation turning out to be more resource intensive than originally expected.

There are various ways in which a cyberorg may attempt to accommodate the needs of its application actors. It may wait till some of the contracts have expired, and not recommit the freed resources; it may wait till its own contract with the host expires, and renegotiate; it may attempt to renegotiate terms or an existing contract; it may count on not all of its client cyberorgs actually utilizing resource availability rates committed to them, and use such resources. Alternatively, a cyberorg may decide to migrate to a different location or use the isolate primitive to create a new cyberorg managing part of its computation and tell it to migrate.

A cyberorg in this simulation has the ability to detect changes in the needs of its actors' computation. This is because a cyberorg maintains a list of active actors which are being awarded ticks, and all create and send requests of actors go through the cyberorg. Not only does a cyberorg know the requirements of its actors, it can also manage them by delaying creations and message deliveries until later.

A simple example of application actors' behavior triggering migration would be for cyberorgs
to maintain thresholds for numbers of active actors, crossing which would trigger an attempt to migrate.

When a cyberorg observes changes warranting migration, it asks the yellow pages service to suggest a suitable alternate host with the needed resources. Having identified a potential host, it attempts to negotiate a contract with it, and when successful, either migrates itself, or migrates a cyberorg created with a subset of the actors and messages intended for them, clients cyberorgs, and a part of its own eCash.

Migration of cyberorgs, as described earlier, is not necessarily a migration across borders of physical machines; migration may simply be to another host cyberorg on the same machine.

**eCash Transfer** Cyberorg migration may also happen in order to transfer eCash to fund another cyberorg in need of eCash to carry out its computations. Typically, a new cyberorg would be created with an amount of eCash, and it would be programmed to migrate and assimilate into the cyberorg requiring additional eCash.

### 6.1.3 Strategies

After honoring their contractual obligations, cyberorgs require strategies for distributing remaining ticks among application actors they are managing. Similarly, actors require strategies for how they utilize ticks they have received; they may prioritize among the number of methods ready for invocation. Additionally, cyberorgs may use specific negotiation strategies, as well as have ways of estimating the appropriate price at which to sell surplus resources or buy needed resources.

### 6.1.4 User Interface

A user can interface with the system using the Actor Foundry shell program called ashell. ashell makes the user the root cyberorg for the system. The user initiates an application run by creating a cyberorg of the desired Cyberorg class with a desired amount of eCash; and next creating an application actor of the desired AppActor class, which would in turn be managed by the cyberorg. Following these creations, the computation progresses simply as the user provides ticks to the cyberorg by sending it tick() messages with an integer parameter specifying the number of ticks being given. Only the user may create eCash or provide ticks.
public void checkForTriggers() {
    if (activeActors.numElements() > 30)
        chores.enqueue("isolate");
    if (activeActors.numElements() < 3)
        chores.enqueue("assimilate");
    if (neededTicks > myTickRate)
        chores.enqueue("migrate");
}

Figure 6.1: checkForTriggers method: Method which checks for thresholds to determine whether it will be advantageous to carry out primitive Cyberorg operations. These requests are inserted into a queue called chores, which is dequeued by the system later.

6.2 Implementing an Application

A system of Cyberorgs is implemented by directly subclassing from Cyberorg and AppActor classes, which are subclasses of the Actor class of Actor Foundry. Additionally, a programmer may customize negotiation protocols and strategies to suit the application.

6.2.1 Cyberorgs

A class of Cyberorgs can be implemented as a subclass of the Cyberorg class. By subclassing from the Cyberorg class, the implemented class inherits a cyberorg’s behavior, which provides a runtime system, which manages the consumption of ticks by application actors, and secures tick resource for their execution from other cyberorgs. The class typically contains one method overriding the checkForTriggers method defined in the Cyberorg class which is invoked periodically to see if a cyberorg primitive needs to be triggered. This method may rely on local information about the cyberorg as well as information about its host cyberorg, which is available from the host cyberorg upon request. Figure 6.1 shows an example implementation of checkForTriggers. This method checks for three conditions: if the number of actors in the cyberorg exceeds a threshold, the isolate primitive is triggered to create a new cyberorg; if the number of actors drops below a threshold, the assimilate primitive is triggered to merges the cyberorg’s contents into its hosting cyberorg; if the number of ticks required is greater than the rate of availability of ticks as stipulated by the contract with the current host, the migrate primitive is triggered.

Additionally, a cyberorg class may also override methods for the behaviors to be triggered when
public void initiateMigrationSequence() {
    // ask yellow page service to find a potential server
    ActorName server = call (myYellowPages, myCurrentRequirements().ticks(),
        myCurrentRequirements().ticksRate());

    // attempt to negotiate with the server
    send (server, "resRequest", self(), myCurrentRequirements().ticks(),
        myCurrentRequirements().ticksRate());

    // if the server is interested in negotiation, contract negotiations
    // commence. If a contract is successfully negotiated, the cyberorg is
    // migrated.
}

Figure 6.2: initiateMigrationSequence method: Method which communicates with the yellow
page service to find a potential host cyberorg, and once found, sends a request for resources to the
cyberorg to begin negotiation for a contract.

public void relocateToServer(ActorName server, Integer ticks, Float rate)
    throws RemoteCodeException {
    send (server, "resRequest", self(), ticks, rate);
}

Figure 6.3: relocateToServer method: Method to be called by an application actor to explicitly
request a migration of the containing cyberorg to a particular cyberorg. This request results in a
resource request being sent to the prospective host, which would initiated a negotiation.

the chore of a particular type is to be carried out. For example, a possible implementation of the
initiateMigrationSequence method for carrying out the migrate chore is shown in Figure 6.2.

In addition to being triggered in response to the state, chores may also be explicitly requested
by application actors. A cyberorg class may override default methods for servicing such requests.
For example, Figure 6.3 shows a method for handling an application actor’s request to relocate the
cyberorg to a different host cyberorg.

Finally, a cyberorg class may override the method for distributing ticks among its application
actors, which - by default - distributes a fixed identical number of ticks to each active application
actor at a time. The class may also override the method containing the default negotiation strategy
which accepts any price for selling ticks so long as it does not represent a loss,\(^2\) and any price for buying ticks that the cyberorg has enough \(e\text{Cash}\) to pay.

### 6.2.2 Application Actors

Implementation of a class of application actors subclasses from the \texttt{AppActor} class. The class defines methods describing behaviors for the application actors as they are defined in subclasses of the \texttt{Actor} class in \textit{Actor Foundry}. However, instead of the usual \texttt{Actor} class primitives of \texttt{create} and \texttt{send}, the programmer uses \texttt{createActor} and \texttt{sendMessage} respectively, with otherwise identical syntax as for class \texttt{Actor}.

For each behavior method, the programmer also includes a method which returns an integer estimating the number of ticks required for the method’s completion given the parameters. By convention, the names of these methods are the behavior method names concatenated with the string “\texttt{Cost}”.

Figure 6.4 illustrates the class definition for an application actor class for the Insurance example discussed previously. A \texttt{ClaimActor} is an actor that receives an insurance claim as well as pointers to three different servers at the time of its creation. These servers are specialized for processing health care, automobile repair, and legal aspects of the claim. On receiving a \texttt{start} method, a \texttt{ClaimActor} creates three different actors to handle the three aspects of the claim,\(^3\) and explicitly requests an isolation of each of these actors into a dedicated cyberorg, followed by migration to the respective servers.

Note the \texttt{startCost} method, which computes the cost of carrying out the \texttt{start} method.

### 6.2.3 Negotiators

Cyberorgs may instantiate given classes of client and server negotiators for negotiating on their behalf or define their own negotiator classes subclassed from the given classes, in which they may customize their negotiation strategies. In either case, negotiators agree on a communication protocol prior to commencing negotiation, and the negotiation behavior must conform to the agreed protocol for the negotiation to successfully conclude.

---

\(^2\)meaning that the price paid for obtaining the ticks is lower than the price at which they are being sold.

\(^3\)Only one specialized actor shown in code segment
public class ClaimActor extends AppActor {

    public ClaimActor (Claim cl, ActorName hS, ActorName rS, ActorName lS) {

        healthServer = hS;
        repairServer = rS;
        legalServer = lS;

        claim = cl;
    }

    public void start () {

        // create actor hA for researching health part of the claim
        ActorName hA = createActor(ClientActor.class,
                                     claim.healthComplexity,claim.healthClaim,healthServer);

        // request isolation of hA into new cyberorg hC with ncash amount
        // of eCash, promised nt number of ticks and ntR rate of tick availability,
        // and ntC and nrC costs per tick and per unit of rate of tick availability

        ActorName hC = (ActorName) call (myCyberorg,
                                           "isolate", hA, ncash, nt, ntR, ntC, nrC);

        // send message to actor hA to request cyberorg hC to migrate to
        // healthServer cyberorg and begin researching the health part of claim
        sendDataActor (hA, "migrateAndExec", healthServer, "research");
    }

    public Integer startCost () {
        return new Integer(2);
    }
}

Figure 6.4: ClaimActor class: An actor that handles an individual claim. It creates actors for handling different aspects of the claim, and requests its containing cyberorg to isolate each of these actors into dedicated cyberorgs. The new cyberorgs migrate to server cyberorgs with specialized resources to handle specific aspects of the claim. Eventually, these cyberorgs would return with actors holding results.
6.2.4 Strategies

Negotiation Strategies

Strategies used by negotiators may be arbitrarily complex. The default strategy is simple: if a cyberorg can afford to pay what is being asked, or if it can afford to sell a resource for the price being offered, it simply accepts; if not, it proposes the best deal it can offer as a counter proposal.

Tick Distribution Strategy

Contractual obligations of a cyberorg, particularly the rate of availability obligations, determine how ticks are to be distributed among client cyberorgs. A cyberorg calculates the number of ticks to award to each client in a single round based on the tick rate commitments it has made. Recall that rate of availability of ticks is in terms of a ration of the total number of ticks being introduced into the system.

Once the obligations to clients have been satisfied, a cyberorg may decide based on local application or resource availability considerations how to distribute the remaining ticks among application actors. The simplest strategy would be to award an equal number of ticks to each application actor. Alternative strategies may award ticks based on how many messages they may have to process, or on how many ticks are required to process those messages. More complex strategies may attempt to identify actors whose completion is being awaited by other idle actors, and award them the most ticks.

This is of particular relevance to speculative computation [48] in parallel And-Or search problems, where it is essential to adjust preferences to favor the most promising of a group of alternative computations being pursued. A cyberorg may adjust the number of ticks provided to actors based on its estimation of promise.

6.3 Implementation Details

6.3.1 CyberOrgs

The CyberOrg class is written as a subclass of the Actor class, and provides a runtime system for managing application actors, and for securing and managing their resource consumption. A
cyberorg has an amount of eCash, a contract with its parent cyberorg, a list of cyberorgs it is
hosting, a list of its local application actors, and a list of undelivered actor messages.

An application actor is an active (application) actor when it has received one or more messages
in its queue which are ready for servicing. A cyberorg is an active cyberorg if it has one or more
active actors or is serving one or more active cyberorgs. An inactive cyberorg or actor informs its
host cyberorg when it becomes active.

Note that cyberorgs, not application actors, are resource-aware; therefore, all actor requests re-
quiring resources go through the cyberorg. These include requests to send local or remote messages,
and to create new local actors.

When a cyberorg receives a request to create a new actor, it decides when to grant the request
based on its resource commitments.4 At the time of creation of a new application actor, the
cyberorg decides what resources to provide it over its active life.

When a message send request is received by a cyberorg, it determines whether the message is
local or remote. If the message is local, it forwards the message to the intended recipient. If
instead the message is remote, it must be inserted into a new cyberorg which would migrate to the
remote location.

Mobility

The notion of remoteness in cyberorgs is not identical to that of a remote machine. Recall that
there may be multiple cyberorgs on a machine. Thus, "migrating" may simply mean migrating to
a cyberorg on the same machine. This involves negotiating a contract with the new host, changing
the myCyberorg pointer to point to the new cyberorg, and letting the new host include the migrating
cyberorg into its list of hosted cyberorgs. Migrating across machine boundaries is potentially more
complicated; however, Actor Foundry’s support for actor migration across machines simplifies the
process: the only difference is that now the migrating cyberorg along with all its hosted cyberorgs
and actors need to invoke the migrate primitive.

4To preserve Actor semantics, a cyberorg may not decide whether - just when - to create a requested actor.
6.3.2 Application Actors

AppActor, the class defining application actors too are implemented as a subclass of the Actor class. In addition to methods representing its behavior, each application actor also has a default method that receives all messages intended for it. This is the method that receives all messages, and directs them to the appropriate methods. Because neither Java provides a mechanism for effective preemption of executing threads, nor Actor Foundry for actors, we need a way of estimating the resource requirements of method executions. We achieve this by having a function for each method, that takes the message parameters used to invoke it, and returns the number of ticks which will be required by the actor to process that message.

6.3.3 Contracts

Contracts between cyberorgs may be arbitrarily complex. This simulation uses simple contracts stipulating the number of ticks to be made available to the client, the rate at which they would be made available, and the costs of both. The cost of the number of ticks is simply represented as the per-tick cost. The cost of a certain rate of availability of ticks is in terms of the available ticks, whether they are or are not used. For example, while a cyberorg is inactive, although it will not be charged for the number of ticks received, it will continue to be charged for the ticks made available to it by its host. This is because the reserved rate of availability could not be made available to another client, resulting in a loss to the host. An alternate approach could be for hosts to be probabilistic in estimating the rates at which resources may be available, taking into consideration that not all clients would be active all the time.

6.3.4 Negotiation

Negotiating the number of ticks is the simplest. A cyberorg knows the total number of ticks it is promised by its host, and the number that remain at any point in time. A cyberorg may use this information to decide what it can offer. The rate of availability of ticks is in terms of the rate at which ticks are being generated at the root. Specifically, the rate is the percentage of ticks that are made available to the root cyberorg. The rate of availability of ticks is guaranteed to a cyberorg so long as it has not exhausted its promised ticks. In other words, the total number of ticks and
the rate at which they are available to a cyberorg determines the duration of the contract. When a cyberorg is negotiating with a prospective client, the rate at which it offers ticks may not exceed the rate at which it is receiving. This is simply because a cyberorg cannot expect to save ticks to offer a better tick rate to its client; unused ticks expire and cannot be saved. Furthermore, a cyberorg may not offer a tick rate beyond the duration of its own contract.

The cost at which to offer ticks may be a complex decision. For our purposes we have implemented a simple scheme in which a server cyberorg accepts any offer to buy its ticks, so long as it is higher than what it is paying to its own host. Similarly, a client cyberorg agrees to buy ticks for a cost it has eCash to pay.

Protocol

The simulation makes it possible for involved parties to agree on one out of a number of available protocols that they would like to use for negotiation. The meta protocol for agreeing on the negotiation protocol is simple and fixed. One cyberorg proposes a protocol; the other either accepts the proposal, proposes an alternative, or declines to negotiate. This communication goes on until either a protocol is accepted or negotiation is declined.

Once a protocol is selected, both the prospective client and the server create special negotiator actors which attempt on their parties’ behalves to negotiate a contract according to which the client may be hosted at the server cyberorg.

A simple example of a negotiation protocol would involve the first party proposing a potential contract, followed by the other party either accepting the proposal, making a counter proposal, or decline. If there is acceptance, the contract is signed; if there is a counter proposal, the participants iterate; if the proposal is declined, the negotiation ends. Figure 6.5 shows the definition of the Negotiator class from which classes ClientNegotiator and ServerNegotiator are subclassed, which negotiate on behalf of a prospective client and a prospective server, respectively.
public class Negotiator extends Actor {
    protected ActorName myCyberorg = null;
    public void counterPropose(String requestId, Contract proposal) {
    }
    public void decline(String requestId) {
    }
    public void accept(String requestId, Contract proposal) {
    }
}

Figure 6.5: Negotiator class: Class from which server and client negotiator classes subclass. These actors act on behalf of cyberorgs they are contained in, to negotiate contracts with other cyberorgs. A negotiator actor is created in response to a request message received by a cyberorg.

6.3.5 Tick Propagation

Ticks to Cyberorgs

The system advances as a result of introduction of ticks. A cyberorg receives ticks from its parent in accordance with the contract between the two. This means that the total number of ticks received by a cyberorg as well as the rate they are received at is determined by the contract. Also determined by the contract is the cost the cyberorg is charged for the ticks.

Ticks to Application Actors

Only active actors are given ticks by a cyberorg. Because we do not have the ability to preempt the execution of an application actor’s method, an actor receiving ticks holds them until it can use them to complete the execution of one of its pending messages. Specifically, an actor keeps collecting ticks until it has enough to proceed with servicing one of its messages. So long as an actor cannot use its current ticks, it returns the control back to the cyberorg. Once an actor has enough ticks to process messages, in what order it services them is a local decision so long as the fairness requirement of Actors semantics is satisfied.
Transfer of Control

Transfer of control in the simulation happens as a result of hosted actors or cyberorgs voluntarily electing to execute or not in response to an invitation to do so by the host cyberorg. Because the simulation is built using Actor Foundry, and is an Actor program, all communication uses asynchronous messages. An active cyberorg receives ticks in the form of a `tick()` message from its host with a value representing the number of ticks being received. Because ticks are received only by active cyberorgs, the recipient of a `tick()` message must either be host to active cyberorgs or have active actors. If it is hosting active cyberorgs, it sends them `tick()` messages with the number of ticks to give them in accordance with their contracts. The ticks that remain after satisfying contractual obligations may be provided to application actors to advance on their computations. An actor receiving `tick()` messages may hold on to the ticks (as described earlier) even though it is active; such ticks may be used when the actor has enough ticks to complete servicing of a message. If an actor has ticks left after servicing all its messages, it returns the remaining ticks to the cyberorg, in the process also informing it that it is no longer active. On receiving this intimation by an actor, if this was the last active actor of the cyberorg, and it is not hosting any active client cyberorgs, it reports to its own host that it too is inactive.

Cyberorgs attempt to optimize delivery of ticks by skipping hosted cyberorgs which are not currently active. This is also a more accurate simulation of how computational resources do become available in real computations. If the recipient is not ready to use a processor cycle or an opportunity to send something over the network, the resource goes unused - in other words, it expires.

6.4 Discussion

The purpose of this prototypical implementation was primarily to illustrate the expressive power of programming constructs based on the CyberOrg model. The system is constructed as an Actor program which uses the programming framework and runtime system of Actor Foundry, a Java library implementation of Actors.

For the purposes of our implementation, we ignored the inner workings of Actor Foundry and assumed it to provide an efficient implementation of Actors. However, we note that this assumption
is not necessarily justified. Actor Foundry is not the most efficient implementation of Actors, and using it to build a runtime system which does fine-grained scheduling poses significant problems.

6.4.1 Atomicity of Methods

How resource consumption is to be controlled offered two design choices, based on whether or not method executions are atomic. If execution of a method may be preempted, one may monitor resources being consumed by an executing method and suspend it before it exceeds its limit. Alternatively, application writers may be expected to honestly provide accurate estimates of resource requirements of all methods. In this case, a method’s execution is initiated only when it is permissible for it to consume resources required to take it to completion.

Because neither Java nor Actor Foundry provides mechanisms for supporting effective control of resources, Cyberorgs use the latter approach. It may be argued that an alternate implementation platform may have been more suitable for implementing Cyberorgs; however, the potential for code mobility offered by virtually universal support for Java across commonly used hardware and operating systems, and Actor Foundry’s support for Actor semantics, meant a cleaner implementation. Furthermore, future versions of Java are expected to provide stronger support for resource control in the form of the upcoming Isolation and Resource Allocation APIs.

A variation of our choice would be to use a source to source transformation of actors’ code so that continuations are captured at points during a method’s execution (such as immediately after an intermediate result has been produced) and the cyberorg is given an opportunity to regain control.

6.4.2 Optimizations

Although performance was not among the salient concerns of this implementation, there were opportunities to optimize which were availed. We discuss the optimizations present in this implementation and further optimizations that may be considered. However, we will limit ourselves to a discussion of optimizing cyberorgs, not the underlying implementation of Actors. For a thorough discussion of optimizations possible for Actor implementations, see [33].

66
Active Vs. Passive

Neither actors nor cyberorgs are always active. In other words, an actor may not always be processing a message, and a cyberorg may not always have active actors and cyberorgs that need scheduling. Given a simple type of contract such as the one we use in this implementation, which stipulates a flat rate of availability of resources over its span rather than specification of a more complex availability pattern, invariably, there are times when cyberorgs are not able to use the resource availability due to them.

Short of mechanisms that over-commit resources - such as the probabilistic commitment described above - significant resources inevitably go unused by cyberorgs who are committed them. We use a combination of scheduling and accounting strategies to partially alleviate this problem. First, a server cyberorg does not hold back on rate of availability of resources when its rate is not entirely committed to its actors and client cyberorgs: clients receive better than promised rate. Second, a server monitors the state of activity of its client cyberorgs and gives them ticks only when they are active. The availability rate so released can be used by the cyberorg as it pleases: it may be used for allowing its actors to advance in their computations, or it may be used to provide better than the committed rate of availability to clients. Additionally, a cyberorg accounts for this, particularly when the released tick rate benefits it, by charging the client for the actual ticks it uses, plus for the rate of ticks for the entire duration of usage.

This scheme may have its drawbacks. For example, a cyberorg no longer knows how long a client may take to consume ticks committed to, and is limited in its ability to negotiate terms with future clients. However, if there are provisions in the contract setting firm expiration times, a server may avail them to suspend the clients’ execution.

Local Migration

Because cyberorg migrations do not always amount to migration over physical nodes, our implementation optimizes migrations to cyberorgs residing on the local machine.
Simple Operations

Tasks such as the handling of remote messages in cyberorgs are very involved. Recall that because only cyberorgs may migrate, a remote message must be enveloped in a cyberorg, possibly along with some eCash, the cyberorg negotiates and migrates to the cyberorg containing the message’s recipient, and there it assimilates. In a real implementation, such processes may be optimized by special handling. The cyberorgs containing the sender and recipient actors can communicate directly and transfer the message for a fee.

Such operations may be further optimized if they involve cyberorgs residing on the same machine.

Negotiation Protocols

In addition to complex negotiations between cyberorgs, there may be cases such as the one involving simple operations which incur fixed costs. In general, there may be negotiation protocols which are simple enough to be dealt with directly between participating cyberorgs rather than be handled by specialized negotiators. This presents another opportunity for optimization.

6.4.3 Overhead

The most significant contributor of overheads in cyberorgs is the cost of context switches resulting from scheduling. Specifically, in a naive implementation of cyberorgs, where a scheduler could be implemented for each cyberorg, creating a hierarchy of schedulers, each scheduling the computations as well as the schedulers corresponding to client cyberorgs they host. The cost of this bureaucracy could be prohibitive.

Suspending and resuming threads in Java is prohibitively expensive, to the extent that any other overheads are dwarfed in comparison. An analysis was carried out to ascertain the scheduling overhead for an implementation of cyberorgs with multilayered scheduling, which uses Java’s suspend and resume primitives for scheduling purposes.

If \( o \) is the overhead resulting from a single suspend-resume, consider the case that all schedules

---

5 Java has deprecated its `suspend` and `resume` primitives because they are prone to errors in variable sharing situations. Because actors do not share variables, it is safe to use these primitives to suspend and resume actors. Upcoming Isolation API from Java is slated to revive the dreaded primitives.
in the hierarchy schedule leaf or scheduler thread under them for a fixed time slice \( t \). Any scheduler suspending a thread under it, would also have to suspend all running threads down to the leaf, and similarly for resuming an alternate thread. In any time period \( t \), there would be \( h \) suspend-resume overheads for a scheduler at each height \( h \), amounting to \((h^2 - h)/2\) overheads. The cost of this per cycle overhead may be bounded by changing the size of the cycle. Specifically, if \( p \) is the percentage of processing time which is acceptable to be consumed by the overhead, we have:

\[
\frac{\frac{h^2 - h}{2}}{t} < p \Rightarrow \frac{h^2 - h}{2} < \frac{pt}{o}
\]

Alternatively, if all schedulers scheduled their threads for time cycles \( mt/h \), where \( h \) is the height of the scheduler, and \( m \) is an integer the inequality would be:

\[
\sum_{h=1}^{m} \frac{ir_i + (1 - r_i)}{m_i} < \frac{pt}{o}
\]

We determined the cost of a single suspend-resume invocation to be roughly 1 millisecond on a Pentium III 1.2 GHz 256 MB machine running JDK 1.3.1 on Windows 2000. Here are the scheduler time slices we obtained for limiting overhead to 3% of the processor time for some sample scheduler configurations:

- \( h = 1, m_1 = 1, p = 0.03, r_1 = 0; t_1 = 33 \) ms
- \( h = 2, m_i = 1, p = 0.03, r_1 = 0, r_2 = 1.0; t_i = 100 \) ms
- \( h = 2, m_i = 1, p = 0.03, r_1 = 0, r_2 = 1.0; t = 66.7 \) ms; \( t_1 = 67 \) ms, \( t_2 = 133 \) ms

The first configuration simply schedules actors without adding an additional layer for cyberorgs; the other have two layers of schedulers. It is obvious that scheduling would need to be significantly coarse grained for multi-layer configurations in order to achieve an acceptably low cost of overhead, making a case for limiting the layers of schedulers to two: the top layer scheduling on the basis of contracts, and the bottom layer on the basis of a cyberorg’s local resource consumption concerns.

Limiting scheduler layers to two does not necessarily mean that cyberorg programs may not have cyberorg hierarchies. What it means is that an implementation should be optimized so as to
collapse the levels to two. Recall the view of cyberorgs in which we saw them as a collection of cyberorgs rather than a hierarchy. Once a cyberorg has negotiated a contract with its host, the contract promises absolute levels of resource availability, meaning that the system scheduler may provide resources to a cyberorg directly rather than through intermediate levels of schedulers.

Finally, we observe that scheduling cost figures so prominently in our discussion of overheads incurred by cyberorgs simply because of inadequate support for suspending and resuming threads in Java, which is ironically the language of choice for implementation of mobile agent systems. Recently, partly inspired by our work on separating resource and functional concerns of resource bounded agents [31] – which led up to development of the CyberOrg model – Sun Microsystems has developed Isolate and Resource Management APIs for Java [11], which promise to be very useful for implementing systems of cyberorgs. The Isolate API disallows object sharing by threads, a concept closer in principle to Actors. This API also re-introduces the previously deprecated suspend and resume primitives, which may now be used without the risks present when threads could share objects. The Resource Management API builds upon the Isolate API to offers an environment very similar to a two level system of cyberorgs. One can expect to see better optimized implementations suspend and resume primitives as they become more widely used.
Chapter 7

Implications and Open Issues

In this chapter, we summarize possible implications of this work on the CyberOrgs model, and identify open issues which should guide future work.

7.1 Scalability

Organization of concurrent computations has a bearing on scalability. Cyberorgs are conceptually organized as a hierarchy. Because computational resources are located in space and at instants of time, change in their ownership too is in space and time. Because a transfer of ownership may be for a duration of time, there is a need to keep track of who the ownership returns to at the end of that duration. Hierarchies are a natural book-keeping tool of representing such a relationship. More significantly, because resource ownerships are transferred by negotiating with the current owner, there is a hierarchy of decisions which lead to the state of resource ownerships at any instant.

There is a well understood cost of communicating control decisions in a hierarchy [41] to where the decisions are enforced. However, the hierarchy of CyberOrgs can be flattened in order to alleviate this cost. Specifically, although there is a hierarchy of control decisions, the decisions can be maintained in a composed form, so that their enforcement uses a flat structure.

7.2 Protection from Viruses

Models of sequential computation coupled with the traditional model for ownership of computational resources implicitly afforded the freedom to ignore resource accounting. Where sequential computation simplified resource sharing between computations, the ownership model addressed
potential for contention for resources by restricting "unauthorized" access. As existence of Internet and popularization of concurrent programming lead to the emergence of open systems, the assumptions that allowed a more informal treatment of resource accounting fail to hold. For example, the popular subscription model for Internet access has permitted SPAM to effectively render electronic mail unreliable because of accidental purges by filtering software. Computer viruses too have exploited the fact that control of resource access is a higher level function rather than a core function of computer systems, in order to proliferate [55].

A tighter control over resource access, coupled with a finer-grained control than the present all-or-nothing model may offer better protection against attacks exploiting resource vulnerabilities. Subscription models for Internet access, for instance, may be harmonized with resource control objectives.

7.3 Real-Time Requirements

Emerging realization of open systems over the Internet makes real-time concerns relevant to a much larger class of computations. Although the enormous amounts of computational resources physically connected by the Internet are potentially available to all computations, there is no coherent basis for sharing of these resource, precluding large scale sharing for general purpose computation. This is so partly because a fundamental assumption that could be traditionally made about execution environments no longer readily holds if such sharing is permitted. With computations freely moving across machines over the network, state of the environment in which computations occur will no longer be locally determined. In other words, the challenge of achieving timely solutions is significantly exacerbated. This, in turn, makes explicit handling of resource control, and consequently, timeliness and quality of service in general, a requirement for realization of open systems.

A distributed computation in an open system may thus be viewed as an attempt to achieve desired properties of states of a network of computers at instants or during time intervals in the future. Examples of these properties include states of availability of solutions to a problem, states of well-behavedness of a distributed operating system, and so on. To reach or maintain such states, computations should execute in environments which allow them to reasonably pursue their
goals. An important characteristic of this property of an environment is the availability of required resources in a predictable manner.

An important implications of this is that resource usage has to be controlled explicitly, and is a matter of negotiation between self-interested resource owners and beneficiaries of computations. Critically, once access has been negotiated, the computation may be carried out in an execution environment that is virtually closed by the negotiated contract.

CyberOrgs offer a multi-agent based model for acquisition and control of resources for distributed applications, which allows applications to execute in an environment of predictable resource availability. The fact that the boundaries of concurrency are independent of resource ownership boundaries enables control of resource consumption of agents as well as ensembles of agents. The model achieves a separation of concerns by representing resource requirements of an application separately from its functionality.

### 7.4 Interaction with Economy

There are no inherent barriers between an economy of computational resources and the existing real-world economy. Particularly, human beings interacting with computer systems are also actors in the real world, allowing an interaction between the economies even though there may not be an explicit relationship between tangible artifacts such as the currencies in use.

With ongoing advances in the fields of electronic commerce and autonomous agents, it is conceivable that future software agents will engage in more complex decision making involving financial transactions. In other words, autonomous agents will carry cash. That combined with an economy of computation promises to expand the sphere of influence of computations. Just as computations are typically initiated by human beings, once software agents are empowered to hold cash and carry out unrestricted financial transactions using them, the agents will be able to initiate activity involving human being. Particularly, software agents will employ physical and human resources to satisfy their goals. Consequently, the objective of human computer interaction would extend beyond pure computation.
7.5 Network Ticks

A cyberorg residing on a physical node having surplus local resources may offer corresponding ticks to its clients. In addition, recall that not all cyberorgs are restricted to a single node. In fact, in a distributed system, there must be cyberorgs which span multiple nodes, the largest cyberorg spanning the entire system. Cyberorgs may span parts of multiple nodes, and parts of the network links between them.\(^1\) For convenience, we will refer to such cyberorgs as network cyberorgs.

7.5.1 Network Cyberorgs

A network cyberorg is any cyberorg that spans more than one physical node with a network connection between them. Network cyberorgs are the only cyberorgs which can offer network resource ticks to other cyberorgs.

Just like cyberorgs that reside on a single node, network cyberorgs too purchase resources from a host cyberorg, which must also span at least those parts of the nodes that the client does. In addition to computation ticks, a network cyberorg also receives network ticks from its host. Network ticks possessed by a cyberorg may be used for communication among application actors over the network spanned by the cyberorg. These ticks are parameterized by the nodes between which they enable communication. Similarly, a network cyberorg may also offer its network ticks to its client network cyberorgs.

Because network cyberorgs reside on multiple hosts, their control structure may need to be more complex than that of other cyberorgs. For example, the cyberorg may maintain client listeners and negotiators at each of the nodes.

Although count of the total number of processor ticks may be reasonably thought to mean the same thing across machines, rates of tick availability pose problems. Recall that we measure rate of availability of ticks in terms of the rate at which the root cyberorg receives ticks. However, when there are multiple nodes (not necessarily cyberorgs), they may be generating ticks and providing them to cyberorgs at different rates. Although a cyberorg that is distributed over different nodes can

---

\(^1\)Cyberorgs represent resource ownership boundaries, not physical boundaries of physical nodes. There may be multiple cyberorgs on a single physical node, multiple physical nodes or parts of them inside a cyberorg. In other words, a computation occurring inside another cyberorg may not just not be on another node; it may be on multiple nodes, some of which may be shared by the two cyberorgs. Distance between two cyberorgs is, therefore, not always clearly defined.
add the rates at which it is receiving ticks from different nodes, the sum serves limited meaningful purpose in determining surplus ticks which may be offered to potential clients.

In a manner similar to network ticks, an attribute identifying the node where the ticks of CPU type are generated may be specified. Given this information, a cyberorg may now keep track of different subtypes of CPU ticks that it receives and the rates at which it receives them.

Transfer of ticks represents delivery of parts of the time-space of resources purchased at the time of signing a contract. The actual resources are obviously located on physical nodes and networks, and their management is done by the nodes. However, their ownership at any time is distributed among cyberorgs. For example, processor time resource is provided by the physical node, but parts of it are owned by different cyberorgs, which may in turn engage in trade involving them. Similarly, a cyberorg may own part of a network resource, however, it may sell part of its resource to another cyberorg. Note that the cyberorg purchasing a network resource will subsequently span multiple nodes.

7.6 Actions at Distance

Although models of concurrent computation have explicitly dealt with communication as well as computation, the objective has been to solve computational problems. Particularly, distribution over time and space have been treated as hurdles to be surmounted. CyberOrgs treat spatial and temporal constraints as an integral part of the problem being solved. The problem, consequently, is not simply to carry out a computation, but to have desired states for points in space at specific times. For example, a telephony application is modeled in part by availability of the voice at a distant location shortly after it is produced, instead of two independent computations remotely coordinating in order to enable a conversation.

7.7 Resource Organization and Discovery

Cyberorgs can report availability of resources held by them to an independent and trusted yellow pages service. This is not necessarily the most efficient mechanism for keeping track of resource availability. For instance, each cyberorg could report surplus resources held by it to its parent
cyberorg, allowing the information to trickle up the network. This too would result in an overhead, but search for a resource location would become logarithmic. Specifically, the root cyberorg would know where specific types of resources are available and direct the potential client down the tree. This would also enable availing resources held by multiple cyberorgs simultaneously.

Note that there may be quantities of different types of resources required by a cyberorg which are not offered by any single cyberorg. This may be true even though the required resources are available on a single host or a group of host spanned by an ancestor cyberorg. In such a case, if the ancestor keeps track of what is available under it, there is a potential for availing resources available through multiple cyberorgs. However, cyberorgs are self-interested, and hence may not be expected to represent their hosted cyberorgs accurately. Specifically, a cyberorg privy to the knowledge of resources available through its clients may choose to be a reseller of the resources after purchasing them from the client, rather than point the buyer to it. Although this may not appear to hurt the sellers, it does add friction to the economy in the form of overheads involved in reselling. In fact, this does hurt the seller by reducing the potential buyers to only their parent.

For this reason, an independent yellow pages service is essential. Cyberorgs would still benefit from reporting resource availability under them to their parents but their ability to sell to multiple buyers will not be compromised. Host cyberorgs may in turn benefit from the information received to offer a larger pool of resources to potential clients, but they would have to compete with the actual owners of those resources. There may be different models for distribution of costs associated with host cyberorgs offering resources owned by their clients. In one model, client cyberorgs may pay a service charge to their host for offering their resources and managing their use following a sale. Alternatively, a host cyberorg may charge a higher cost from buyers for reselling resources purchased from its clients. In either case, it is important to note that considering that it is the host that provides resources to client cyberorgs, redirecting the same resources to a different client may reduce rather than increase its costs.\(^2\) This would mean that reselling - one way or another - would add inefficiency to the economy and result in inflation.

\(^2\)A potential increase in cost may be because of added bookkeeping of hosting a cyberorg.
7.8 Resource Types and Locations

There are interesting ways in which cyberorg ticks may be extended. For instance, ticks may be typed to represent different types of resources. Ticks may be sub-typed based on which cyberorg they are located on, and they may have attributes based on their quality. Consequently, a simple count of resources is no longer sufficient to represent the resources required to take a step forward toward the goal.

Network Ticks Systems of cyberorgs spanning distributed computers present the challenge of representing network resources for enabling communication. Recall that a cyberorg may be distributed over a number of physical nodes. Communication between concurrent computations distributed over a number of machines would require network ticks. Network ticks pose interesting representation challenges. Specifically, the type of a network tick would include identification of the nodes being connected. Critically, this requires an explicit representation of physical nodes being connected. Network ticks to connect to a computation on a remote host may be held by a cyberorg which does not host any computation on the remote host.

There may be ticks of different types (possibly on multiple cyberorgs) which are required simultaneously to make progress. Computations requiring multiple types of ticks simultaneously to take the next step may be modeled in CyberOrgs by a vector of different types of ticks preceding the reduced expression. Recall that a tick is defined in time and space, meaning that it is available only in the cyberorg where it exists, and it expires when the time of its existence has passed. As a result, cyberorgs may receive multiple ticks of different types which all exist at the same time. For example, multiple processor ticks may represent availability of multiple processors, multiple network ticks may represent availability of multiple network connections between different cyberorgs. Note that cyberorgs physically co-located do not require network ticks for communication between their hosted computations.
Chapter 8

Conclusion

There are a variety of advances in distributed systems research which have important implications for Cyberorgs.

A number of approaches are being used to acquire resources for solving challenging problems. At one end of the spectrum are technologies which take advantage of social networks, and whose success or failure depends on social acceptability. Increasingly, massively parallel computation-intensive divide-and-conquer problems are being solved using resources volunteered by owners of personal computers. A recent example was that of the University of Alberta CISS project [49] in which 1380 processors across Canadian cities worked for 24 hours in November 2002, to solve a complex chemistry problem that would have otherwise taken years to complete. Similarly, Internet search leader Google has recently launched Google Compute, an initiative to harvest idle computer cycles of users who have installed their toolbar.

Peer-to-peer file sharing systems have been steadily gaining popularity and increasing the comfort level average computer owners feel about allowing access to their computers. Electronic Commerce, though still a cause of apprehension for many, is nevertheless on the rise.

At the other end of the spectrum are infrastructure initiatives aimed at solving large problems with commercial interests. Computational grids are becoming a popular way of pooling resources among large resource owners. After Akamai’s success in selling network and storage capacity for web service, IBM has announced a $10 billion Utility Computing effort.

Volunteering of resources, just like advertisement supported services over the Internet, does not afford long-term sustainability. There are significant costs associated with computation - rising need for electric power to name one - which ought to be important considerations when talking
about volunteering idle resources. Furthermore, free availability of resources leads to wasteful use, or worse, abuse or misuse of the resources. Network attacks in the recent past have taken advantage of unbounded resource availability in terms of numbers of email messages, amount of data transfer, etc. In general, without bases for putting bounds on the resource consumption behaviors of agents, it is not possible to offer an execution environment to applications which is reliable.

Although research in Grid Computing is developing technologies which will be useful in supporting large distributed computations over peer networks, the solutions are typically too low level to be readily usable for system design.

Importance of addressing these concerns is being increasingly recognized by a variety of research communities. The annual Real-Time and Embedded Technology and Applications Symposium (RTAS) has introduced a new stream in Quality of Service in Open Systems, recognizing the centrality of resource control for computation in open systems. The annual Autonomous Agents and Multi-Agent Systems conference featured a keynote address on convergence of Grid Computing and Multi-Agent Systems in 2004. With emergence of low-level technologies such as are being pursued by the Grid Computing community, as well as stronger support for resource control and object isolation in existing languages such as Java, there will be greater opportunities for building reliable infrastructures for peer-to-peer sharing of resources over Internet to support multi-agent computations.

In this context, CyberOrgs offer a way of deploying large multi-agent applications over networks of privately owned computers. The CyberOrg model separates functional concerns of an application from its resource usage concerns. By modeling quantities and costs of resources separately, CyberOrgs model ownership of resources. The model offers simple primitives which may be used for resource-oriented interactions between self-interested cyberorgs. Finally, it provides a predictable environment for computations to be coordinated in for timely completion.

In this thesis, we have identified the opportunities afforded and the challenges posed by emergence of Internet as a network of millions of privately owned computers, for large scale distributed computing. We have introduced the Cyberorg model for complex resource-bounded mobile agents and presented transition rules representing its operational semantics by instantiating it with Actors as a computation model. We have described an implementation of CyberOrgs as an Actor pro-
gram implemented over Actor Foundry – a Java library implementing Actor semantics – and have illustrated the expressive power of the model by showing how Cyberorg programs may be written.
References


[60] N. Venkatasubramanian and C. L. Talcott. Reasoning about Meta Level Activities in Open


on Agent Systems and Applications and Third International Symposium on Mobile Agents

[64] T. Wagner, A. Garvey, and V. Lesser. Criteria directed task scheduling. Journal for Approxi-
mate Reasoning—Special Scheduling Issue, page 91.

Vita

Nadeem Jamali was born on June 6, 1969 in Glasgow, Scotland. He grew up in Karachi, Sindh, where he attended the Habib Public School and the St. Patrick’s High School for primary and secondary schooling, and graduated from the Dayaram Jethmal Science College with a Higher Secondary Certificate in 1986.

In the Spring of 1987, Nadeem joined the University of Karachi’s Foundation for Advancement of Science and Technology (FAST) Center of Excellence in Computer Science. In December 1989, Nadeem graduated with honors and received a Bachelor of Computer Science.

In January 1991, Nadeem enrolled in the Master of Computer Science program at the Technical University of Nova Scotia. He received the 1990 Scholarship at the Technical University of Nova Scotia. Nadeem completed his Master’s degree in December 1992, under the guidance of Dr. Ping Liang, with a thesis titled “Multi-layer Feedforward Neural Networks with Nonlinear Synapses.”

In January 1993, Nadeem enrolled in the Ph.D. program at the Department of Computer Science at the State University of New York at Buffalo. While at SUNY at Buffalo, Nadeem served as a Teaching Assistant in the Department of Computer Science. In the summer of 1994, Nadeem served as an Instructor for an introductory Computer Science course.

In January 1995, Nadeem transferred to the Ph.D. program at the Department of Computer Science at the University of Illinois at Urbana-Champaign. While at University of Illinois, Nadeem served as a Research Assistant in the Open Systems Laboratory under the direction of Professor Gul Agha. He served as a Visiting Lecturer in the Department in the Spring of 2001. Nadeem also served as a Teaching Assistant in the Department of Computer Science for a number of terms. In 1998, Nadeem was awarded the Kodak Fellowship from the Eastman Kodak Company. During the summer of 1999, Nadeem served as a Co-operative Intern at the Eastman Kodak Company in Rochester, NY.
Since defending his dissertation in December 2002, Nadeem has been an Assistant Professor at the University of Saskatchewan, Canada.