# CyberOrgs: A Model for QoS Management in Open Systems

Nadeem Jamali
Department of Computer Science
University of Saskatchewan
57 Campus Drive
Saskatoon, SK, S7N 5A9, Canada
jamali@cs.usask.ca

Gul A. Agha
Department of Computer Science
University of Illinois
201 N. Goodwin Avenue
Urbana, IL 61801, USA
agha@cs.uiuc.edu

## Abstract

*Realizing open systems over the Internet requires explicit treatment of resource control, and consequently, timeliness and quality of service. We describe CyberOrgs, a hierarchical model for resource sharing between and within object-oriented concurrent computations executing over a network of owned computers. We introduce programming abstractions derived from the model through a prototype implementation of CyberOrgs as an Actor program. An approach is described for implementing CyberOrgs efficiently by flattening the hierarchical schedule in runtime.*

## 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. An important characteristic of this property of an environment is the availability of required resources in a predictable manner.

Since the emergence of Internet, enormous amounts of computational resources are physically connected by the Internet, which are potentially available to all computations. What is lacking, however, is a basis for sharing of these resources, which are typically independently owned. There are multiple sources of unpredictability in a computational environment. A computation's resource requirements may change in an unpredictable way (for instance, because requirements are a function of partial results). But more interestingly, other computations may be competing for available resources, in effect threatening its prospects of receiving what it requires. In an open system where computations may enter or exit the system at any time, this is a typical scenario.

Multi-agent systems may solve problems by competing or collaborating in a problem space. Additionally, given bounded resources, they may also compete or collaborate in the space of a network of resources. To address multi-agent interactions in a resource space, we draw inspiration from populations of biological organisms and commercial organizations. 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 strictly coordinated manner. Similarly, where commercial organizations compete with each other for financial resources, the decision about how to distribute the secured resources among 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.

Ether [6] 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 an Actor language, Acore [8]. Sponsor actors accompanied computation requests, and they carried ticks that could be used in processing a request. Using a similar scheme, in Telescript [13], processes were awarded funds in terms of "teleclicks" which they were supposed to use to accomplish their results.

The Quantum [10] framework is the most relevant to our approach. Motivated by the need to manage limited resources that are 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 ex-

hausted 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 [11] to handle management of distributed and multi-type resources, which does address migration.

## 2. CyberOrgs

CyberOrgs ("cyber organizations") is a hierarchical model for resource sharing between multi-agent computations executing over a network of computers, where agents may migrate in order to avail remotely located resources. Each cyberorg represents a boundary around a concurrent computation and resources committed to its execution. Specifically, a 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 cyberorgs. A cyberorg may not create eCash *ex nihilo*.

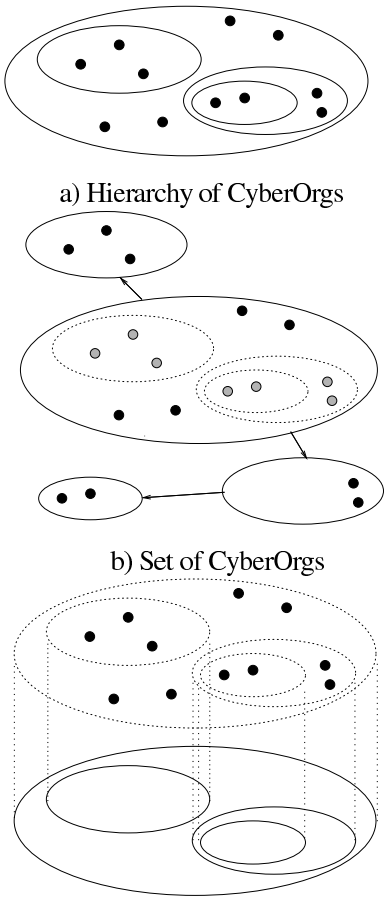
CyberOrgs may be seen as mobile traders and managers of computations and computational resources. Each cyberorg manages sets of computations and resources, hosts other cyberorgs, and owns eCash with which it may purchase resources from other cyberorgs.

CyberOrgs organize resource space as a tree. 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 which 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.

### 2.1. Hierarchy of CyberOrgs

Each cyberorg receives its resources from its host cyberorg, which creates a hierarchy. Figure 1a shows how computations - represented by black dots - and cyberorgs - represented by ellipses - may be 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



a) Hierarchy of CyberOrgs

b) Set of CyberOrgs

c) Transparent Middleware

**Figure 1. CyberOrg Views**

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. 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 by themselves; younger chil-

dren may be bought things by parents; a child yet to be born is physically sustained by the mother.

### 2.2. Collection 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 1b shows this view. 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.

### 2.3. Transparent Middleware

CyberOrgs may also be viewed as a middleware in a facilitating role. As Figure 1c 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.

## 3. 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.

### 3.1. Actors

Actors [1, 4] 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.

It is possible to extend any sequential language with actor constructs. For example, the call-by-value $\lambda$-calculus is extended in [3].

Agents are naturally modeled by the Actor formalism. In fact, many implementations of agents have typically been implementations of actor systems. An actor is autonomous and persistent. Actors are inherently concurrent and autonomous enabling efficient parallel execution [5] and facilitating mobility [2].

Ren and Agha have developed a language for describing and implementing real-time constraints for actor computations [12]. 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.

Consider a variation of the Producer/Consumer problem where the produced object must lie in the buffer for a certain amount of time before being removed by the consumer. A *Time Constrained Producer/Consumer* problem can be implemented by writing writing the code for the usual Producer Consumer problem without explicitly considering the

time constraint (Figure 2). Then, separately an RTSynchronizer can be declared with the time constraint that would prevent the Consumer's `get` request to be delivered until the required amount of time has elapsed. The declaration are translated into the correct scheduling of actors, if such a translation is feasible.

### 3.2. Instantiating CyberOrgs with Actors

In the CyberOrgs model, we represent an actor by counts of resources it requires for processing received messages, functions of messages which return revised counts of resources which will be required following reception of messages, and finally, the messages that would be released. Because the internal representation of messages does not have a bearing on the model, we do not model it. We simply note that at specific points in the execution of an actor, messages are released into the cyberorg.

*Ticks* serve as the unit of resource. Every computation requires a certain number of ticks to complete.

Hierarchy of cyberorgs is not represented by syntactic containment. Information about hosting relationships is maintained in a name table. Containment of actors within cyberorgs is also represented in a similar fashion.

A system of cyberorgs is represented as $\langle\langle \Gamma | \mathcal{M} | \mathcal{C} | \Theta | \mathcal{D} \rangle\rangle$, where $\Gamma$ is the set of cyberorgs in the system, $\mathcal{M}$ is a name table which maps cyberorgs and actors to cyberorgs containing them, $\mathcal{C}$ is the set of contracts between client cyberorgs and their host cyberorgs, $\Theta$ are directed ticks in the system, that is, ticks intended for client cyberorgs and actor, and finally $\mathcal{D}$ is a matrix which keeps track of distances between cyberorgs and actors.

```
Producer(){
  methods:
    put();
    other();
}

Consumer(){
  methods:
    get();
    other();
}
```

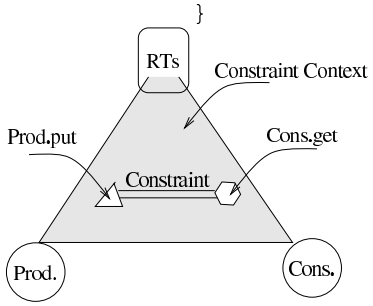



**Figure 2. Producer/Consumer with Time-Bounded Buffer**

An instantaneous snapshot of a cyberorg - a cyberorg configuration - is represented as $[\alpha, \mu, \$, \chi]_{c_i}^{\xi,\omega}$, where $\alpha$ represents the set of actors contained within it, each actor represented by its state (*active* or *inactive*), the ticks it must receive before completion, interspersed with message releases; $\mu$ is a set of actor messages with local or remote recipients; $\$$ is the amount of eCash in the cyberorg; $\chi$ is the list of *chores* (that is, cyberorg primitives) to be executed by 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.

**3.2.1. 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 its chores or 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 or used for advancing on the cyberorg's chores.

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 for completion of $a$ from $n$ to $n-1$.

$$\langle\langle [\{[\![n]\!]_a, \alpha\}, \mu, \$, \chi]_{c_1}^{\xi,\omega}, \Gamma | \mathcal{M} | \mathcal{C} | t(c_1), \Theta | \mathcal{D} \rangle\rangle$$
$$\longrightarrow \langle\langle [\{[\![n-1]\!]_a, \alpha'\}, \mu', \$, \chi]_{c_1}^{\xi',\omega'}, \Gamma | \mathcal{M} | \mathcal{C} | \Theta | \mathcal{D} \rangle\rangle$$
$$T_{c_1}(st(c_1), co(c_1)) = a$$

As a result of delivery of this tick to actor $a$, new actors may be created in the cyberorg, changing the list of other actors from $\alpha$ to $\alpha'$. Similarly, the resource offerings and requirements of the cyberorg could also have changed.

When the tick delivery decision process determines that the tick is to be used for making progress on the cyberorg's chores, number $n$ representing the number of ticks required before the next chore may be activated is decremented. Note

that the chore list is a sequence of pairs representing number of ticks and a chore, the number of ticks meant to represent the cost of executing the chore to follow.

$$\langle\langle [\alpha, \mu, \$, [\![n]\!].\chi]_{c_1}^{\xi,\omega}, \Gamma|\mathcal{M}|\mathcal{C}|t(c_1), \Theta|\mathcal{D}\rangle\rangle$$
$$\longrightarrow \langle\langle [\alpha, \mu, \$, [\![n-1]\!].\chi]_{c_1}^{\xi',\omega'}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$$
$$T_{c_1}(st(c_1), co(c_1)) = chore$$

A chore is activated once the ticks preceding it have decremented to zero:

$$\langle\langle [\alpha, \mu, \$, [\![0]\!].\chi]_{c_1}^{\xi,\omega}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$$
$$\longrightarrow \langle\langle [\alpha, \mu, \$, \chi]_{c_1}^{\xi',\omega'}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$$

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 ticks 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 $c_1$.

$$\langle\langle [\alpha_1, \mu_1, \$_1, \chi_1]_{c_1}^{\xi_1,\omega_1}, [\alpha_2, \mu_2, \$_2, \chi_2]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}|\mathcal{C}|t(c_1),$$
$$\Theta|\mathcal{D}\rangle\rangle \longrightarrow \langle\langle [\alpha_1, \mu_1, \$_1 + \Delta, \chi_1]_{c_1}^{\xi_1',\omega_1'},$$
$$[\alpha_2, \mu_2, \$_2 - \Delta, \chi_2]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}|\mathcal{C}|t(c_2), \Theta|\mathcal{D}\rangle\rangle$$
$$\text{if } < c_1, c_2 > \in \mathcal{M}, T_{c_1}(st(c_1), co(c_1)) = c_2$$
$$\text{where } \Delta = cost_t(st(c_1), co(c_1, c2))$$

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

$$\langle\langle [\alpha, \mu, \$, \chi]_{c_1}^{\xi,\omega}, \Gamma|\mathcal{M}|\mathcal{C}|t(c_1), \Theta|\mathcal{D}\rangle\rangle \longrightarrow \langle\langle [\alpha, \mu, \$, \chi]_{c_1}^{\xi',\omega'},$$
$$\Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle, T_{c_1}(st(c_1), co(c_1)) = \phi$$

**3.2.2. Creation and Absorption** As illustrated in Figure 3a, 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 aught 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.

$$\langle\langle [\alpha_1, \mu_1, \$_1, isolate(\alpha_2, \mu_2, \$_2, \chi_2, \xi_2, \omega_2).\chi_1]_{c_1}^{\xi_1,\omega_1},$$
$$\Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle \longrightarrow \langle\langle [\alpha_1 - \alpha_2, \mu_1 - \mu_2, \$_1 - \$_2, \chi_1]_{c_1}^{\xi_1',\omega_1'},$$
$$[\alpha_2, \mu_2, \$_2, \chi_2]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}'|\mathcal{C}'|\Theta|\mathcal{D}\rangle\rangle$$
$$\alpha_2 \subset \alpha_1, \mu_2 \subset \mu_1, \$_2 \leq \$_1, c_2 \text{ fresh}$$

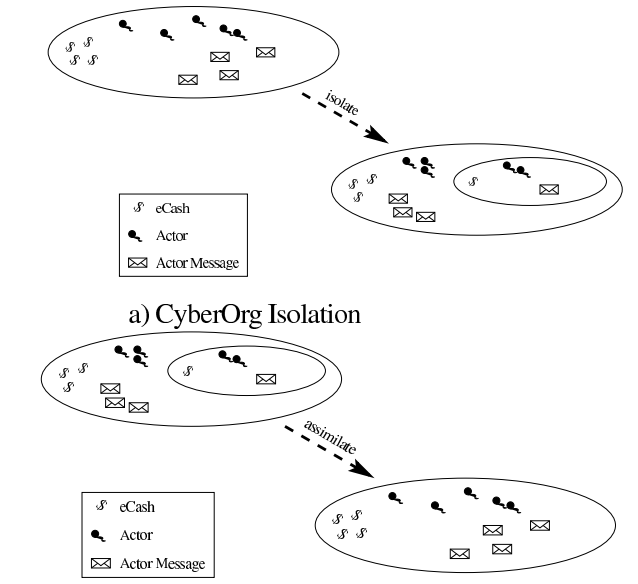



**Figure 3. Creation and Absorption**

As shown in Figure 3b, a cyberorg disappears by assimilating into its host cyberorg using the `asmlt` 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.

$$\langle\langle [\alpha_1, \mu_1, \$_1, \chi_1]_{c_1}^{\xi_1,\omega_1}, [\alpha_2, \mu_2, \$_2, asmlt.\chi_2]_{c_2}^{\xi_2,\omega_2},$$
$$\Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$$
$$\longrightarrow \langle\langle [\alpha_1 \cup \alpha_2, \mu_1 \cup \mu_2, \$_1 + \$_2, \chi_1.\chi_2]_{c_1}^{\xi_1',\omega_1'},$$
$$\Gamma|\mathcal{M}'|\mathcal{C}'|\Theta|\mathcal{D}\rangle\rangle$$
$$\text{if } < c_1, c_2 > \in \mathcal{M}, \nexists c_3 \text{ such that } < c_2, c_3 > \in \mathcal{M}$$

**3.2.3. Mobility** A cyberorg may realize that its resource requirements have exceeded what is available by its contract with the host cyberorg. This triggers its attempts to migrate:

$$\langle\langle [\alpha, \mu, \$, \chi]_{c_1}^{\xi,\omega}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$$
$$\longrightarrow \langle\langle [\alpha, \mu, \$, [\![n]\!].search(\xi).\chi]_{c_1}^{\xi,\omega}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$$
$$\text{if } \xi > av(co(c_1, c_2)) \text{ where } < c_1, c_2 > \in \mathcal{M}$$

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

1 Migration of a part of a cyberorg's computation would require isolation first.

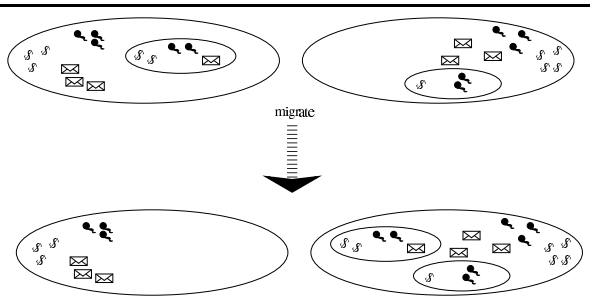



**Figure 4. cyberorg Migration**

cyberorgs which may offer needed ticks for an acceptable price.

$\langle\langle [\alpha_1, \mu_1, \$_1, search(\xi_1).\chi_1]_{c_1}^{\xi_1,\omega_1}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$
$\longrightarrow \langle\langle [\alpha_1, \mu_1, \$_1, [\![n]\!].negotiate(C).\chi_1]_{c_1}^{\xi_1,\omega_1}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$
where $C$ is the set of cyberorgs found $\{c_k, c_{k+1}, \cdots, c_l\}$, possibly including $c_1$

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[2].

$\langle\langle [\alpha_1, \mu_1, \$_1, negotiate(C).\chi_1]_{c_1}^{\xi_1,\omega_1}, [\alpha_2, \mu_2, \$_2, \chi_2]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$
$\longrightarrow \langle\langle [\alpha_1, \mu_1, \$_1, [\![n]\!].migrate(c_2).\chi_1]_{c_1}^{\xi_1,\omega_1}, [\alpha_2, \mu_2, \$_2, \chi_2]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}|\mathcal{C}'|\Theta|\mathcal{D}\rangle\rangle$
if $C \neq \phi$, where $c_2 \in C = \{c_k, c_{k+1}, \cdots, c_l\}$ such that $\xi_1 < \omega_2, \mathcal{C}' \supseteq \mathcal{C}$

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:

$\langle\langle [\alpha_1, \mu_1, \$_1, negotiate(C).\chi_1]_{c_1}^{\xi_1,\omega_1}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$
$\longrightarrow \langle\langle [\alpha_1, \mu_1, \$_1, \chi_1]_{c_1}^{\xi_1',\omega_1}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$
if $C = \phi \vee \nexists c_i \in C = \{c_k, c_{k+1}, \cdots, c_l\}$ such that $\xi_1 < \omega_i$

2 A migrating cyberorg may not exist at the time of negotiation; it may be created following a successful negotiation.

3. **Migrate** to the selected host. If a contract has been successfully negotiated, a client client can relocate to the host using the migrate primitive as shown in Figure 4.

$\langle\langle [\alpha_1, \mu_1, \$_1, migrate(c_2).\chi_1]_{c_1}^{\xi_1,\omega_1}, [\alpha_2, \mu_2, \$_2, \chi_2]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$
$\longrightarrow \langle\langle [\alpha_1, \mu_1, \$_1, \chi_1]_{c_1}^{\xi_1',\omega_1'}, [\alpha_2, \mu_2, \$_2, \chi_2]_{c_2}^{\xi_2',\omega_2'}, \Gamma|\mathcal{M}'|\mathcal{C}|\Theta|\mathcal{D}'\rangle\rangle$
$if \{co(c_1, c_2)\} \in \mathcal{C}$, where $\mathcal{M}'$ reflects the change in location of $c_1$

where $\mathcal{D}'$ 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:

$\langle\langle [\alpha_1, \mu_1, \$_1, mig(c_2).\chi_1]_{c_1}^{\xi_1,\omega_1}, [\alpha_2, \mu_2, \$_2, \chi_2]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle \longrightarrow \langle\langle [\alpha_1, \mu_1, \$_1, \chi_1]_{c_1}^{\xi_1',\omega_1'}, [\alpha_2, \mu_2, \$_2, \chi_2]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\rangle$
$if \{co(c_1, c_2)\} \notin \mathcal{C}$

cyberorgs progress 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 and chores 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.

Furthermore, awarding ticks one at a time allows management of rates of use of resource. cyberorgs may offer absolute rates of availability of resources 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.

Distribution of resources a local decision but the decision process is not shown. This can be used for resource-oriented coordination.

If a tick is received by a cyberorg when it does not have a computation, chore 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.

## 4. Properties

Preliminary analysis of the model has found cyberorgs to hold the following properties.

**Proposition 4.1** *The amount of eCash in the system remains constant.*

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

**Proposition 4.2** *If no new ticks are inserted into the root cyberorg, the system will eventually become dormant*

**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 forwarded to the chore list which consumes it and advances one step. If the chore's counter has decremented to zero, the chore is carried out. Further progress requires another tick.
- 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 the chore list which consumes it and advances one step. If the chore's counter has decremented to zero, the chore is carried out. Further progress requires another tick.
- or forwarded to a client cyberorg, in which it is consumed, and the cyberorg stops.
- or expires because the cyberorg cannot use it.

**Proposition 4.3** *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 quantums,*[3] *then the system will eventually become dormant.*

**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.
*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, bringing the cyberorg to a dormant state in either case.

Work is ongoing on specifying CyberOrgs in the Maude rewriting logic system [9]. Maude is a reflective language and system for giving executable semantics to languages and models of concurrency using rewriting logic, which is a particularly suitable logic for representing concurrent object-oriented computations. The executable specification being developed for CyberOrgs will be used for rigorously studying the model's properties.

## 5. Prototype

A prototype cyberorg implementation has been built using Actor Foundry [7], a library of Java classes supporting Actor functionality. Actor programs may be written and executed in a run time that supports operational semantics of the Actor model.

CyberOrgs are implemented as a library of Actor Foundry code. Specifically, cyberorgs are implemented as an Actor program using the Actor Foundry library. CyberOrgs and AppActors are implemented as subclasses of the Actor class to provide a runtime system for managing application actors and for securing and managing resources.

3 Similar in principle to impossibility of Zeno machines: one cannot do infinite amount of computation with finite amount of eCash

```
public class Traveler extends Cyberorg {
 public void migrateTrigger() {
  if (myCurrRequirements() >
          myCurrService())
      initiateMigrationSequence();
 }

 public void initiateMigrationSequence() {
  ActorName server =
    call (myYellowPages,
    myCurrRequirements().ticks(10),
    myCurrRequirements().ticksRate(10));
  send (server, "resRequest", self(),
    myCurrRequirements().ticks(10),
    myCurrRequirements().ticksRate(10));

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

**Figure 5. Traveler CyberOrg**

A system of cyberorgs is implemented by directly subclassing from CyberOrg and AppActor classes. There are three important components of a CyberOrg system:

**CyberOrgs.** A class of cyberorgs is implemented as a subclass of the CyberOrg class. By subclassing from the CyberOrg class, a programmer's class inherits a cyberorg's behavior, which provides a runtime system for managing the "tick" consumption of application actors, and securing "ticks" resource for them from other cyberorgs. The subclass typically contains methods implementing triggers for cyberorg primitives, relying on local information about the cyberorg as well as information about its host cyberorg, which is available from the host upon request. Furthermore, cyberorg code may override the default methods for distributing ticks among its application actors or the method containing the default negotiation strategy. Figure 5 shows implementation of a class of cyberorgs. A Traveler cyberorg's migration is triggered by its resource requirements exceeding available resources. When this happens, it obtains a name from the yellowpages server of a potential host cyberorg to migrate to. It sends a query to the potential host asking for a quote for needed resources, following which negotiation may commence.

**Application Actors.** Implementation of a class of application actors subclasses from the AppActor class. The class defines methods describing behaviors for the AppActor as for any other Actor subclass. However, instead of the usual Actor class primitives of `create` and `send`, the programmer uses `createActor` and `sendMessage` respectively, with otherwise identical syntax as for class 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 "`Cost`".

**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.

### 5.1. An Efficient Implementation

Work is ongoing to develop an efficient implementation of CyberOrgs in Java. An important hurdle in efficiently implementing cyberorgs is the model's hierarchical structure. A naive implementation using a tree of schedulers incurs a scheduling overhead proportional to the tree's depth for suspension or resumption of a cyberorg high in the tree. This overhead can be controlled by scheduling cyberorgs for time slices proportional to their respective heights in the tree. However, this leads to an unacceptably coarse granularity of control.

Our approach is to flatten the hierarchical schedule at run-time. In other words, instead of launching a new scheduler for each cyberorg, cyberorgs' internal schedules are merged into a single flat schedule which preserves their respective allocations.

## 6. Conclusions

Agents sharing an execution environment invariably compete for available resources, possibly in ways impacting global performance of a multi-agent application. CyberOrgs offer a model for acquisition and control of resources for multi-agent 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. Case studies are needed to assess the effectiveness of Cy-

berOrgs as a programming framework for resource control.

## References


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

[2] G. Agha and N. Jamali. Concurrent programming for distributed artificial intelligence. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to DAI.*, chapter 12. MIT Press, 1999.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1996. to appear.

[4] Gul Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[5] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, 1995.

[6] William A. Kornfeld and Carl Hewitt. The scientific community metaphor. *IEEE Transactions on System, Man, and Cybernetics*, 11(1):24–33, January 1981.

[7] Open Systems Laboratory. The Actor Foundry: A Java-based actor programming environment. *Available for download at ⟨http://www-osl.cs.uiuc.edu/foundry⟩*.

[8] Carl Manning. ACORE: The design of a core actor language and its compiler. Master's thesis, MIT, Artificial Intelligence Laboratory, August 1987.

[9] J. Meseguer. A logical theory of concurrent objects and its realization in the maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*. The MIT Press, 1993.

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

[11] Luc Moreau and Christian Queinnec. Distributed and multi-type resource management. In *ECOOP'02 Workshop on Resource Management for Safe Languages*, Malaga, Spain, June 2002.

[12] S. Ren and G. Agha. RTSynchronizers: Language support for real-time specifications in distributed systems. In *Proceedings of ACM SIGPLAN 1995 Workshop on Languages, Compilers, and Tools for Real-time Systems*, pages 55–64, 1995.

[13] J. E. White. Telescript Technology: The Foundation for the Electronic Marketplace. Technical report, General Magic Inc., Mountainview, CA, 1994.