Scalable Hierarchical Coordination of Multi-Agent Resource Usage

Nadeem Jamali and Xinghui Zhao

Department of Computer Science
University of Saskatchewan
57 Campus Drive
Saskatoon, SK, S7N 5A9, Canada
{n.jamali, x.zhao} @usask.ca

Abstract. Scalable coordination is a key challenge in deployment of massively multi-agent systems. Resource usage is one part of agent behavior which naturally lends itself to abstraction. CyberOrgs is a model for hierarchical coordination of resource usage by multi-agent applications in a network of peer-owned resources. Programming constructs based on the CyberOrgs model allow resource trade and reification of control while maintaining a separation between functional and resource concerns of applications. An operational semantics of CyberOrgs is presented. Expressive power of programming constructs based on CyberOrgs is illustrated with examples.

Hierarchical control presents challenges in scalability. However, when CyberOrgs are used, some types of resource coordination are amenable to efficient implementation. Hierarchical scheduling for processor time, for instance, can be implemented scalably by efficiently converting the hierarchical schedule into a flat schedule on the fly. This mechanism can be generalized to achieve scalable coordination of some other resource types. An implementation of such a scheduler for hierarchical scheduling of cpu usage is described. Experimental results are presented which demonstrate scalability of the approach.

1 Introduction

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 an agent's decision about the action to take next depends on actions taken by other agents, *coordination* between the agents is required to achieve optimal results [5]. Not only is coordination recognized as a key concern in distributed computing [4], it has also been argued that computation and coordination are separate and orthogonal dimensions of all useful computing [6], necessitating coordination to be addressed explicitly.

Computations sharing an execution space inevitably compete for the resources in that space. In an *open system* [7], there may be both *logical* and *resource* dependencies [5] between agents, with the resource dependencies sometimes leading to logical dependencies. Unrestricted competition for resources between agents collaborating to

achieve a shared goal may hamper progress toward the goal. Coordinating resource access by agents is hence critical to reducing uncertainty and enabling agents to make control decisions for the best global performance [9].

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 k, if we begin with n such agents, at the end of the end of the cth cycle of the scheduler, the number agents is $(n(1 - p/k))^c$, which represents an exponential growth.

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 bookkeeping 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.

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.

Ether [11] 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* [13]. *Sponsor actors* accompanied computation requests, and they carried ticks that could be used in processing a request. Using a similar scheme, in *Telescript* [16], processes were awarded funds in terms of *teleclicks* which they were supposed to use to accomplish their results.

The Quantum [14] framework is the most relevant to our work on CyberOrgs. Motivated by the need for managing finite resources shared by multiple computations, Quantum models resources as *energy* which computations require for execution. Com-

putation tasks are contained in *groups* which also serve as *tanks* of energy. Groups are hierarchical, 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 [15] to handle management of distributed and multi-type resources, which does address migration in a limited manner.

2 CyberOrgs

CyberOrgs [8] 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 one or more multi-agent computations (to be referred to as computations contained in the cyberorg), and distributed resources available to it (to be referred to as resources owned by the cyberorg) for carrying out its computations or for resale. Cyberorgs act as principals in a market of resources, where they can buy or sell resources among themselves using *eCash* in a shared currency. To avail themselves of acquired resources, cyberorgs migrate in the space of cyberorgs.

CyberOrgs treat computational resources as being defined in time and space. Sale of a resource is represented by a *contract* stipulating availability of resources to the buyer for a cost. 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.

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.

2.1 Actors

Actors [1] 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 λ-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 [10] and facilitating mobility [2].

2.2 Instantiating CyberOrgs with Actors

An instantaneous snapshot of a system of cyberorgs is represented by $\langle \Gamma | \mathcal{M} | \mathcal{C} | \Theta | \mathcal{D} \rangle$. Γ 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, Θ 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 $\mathcal{D}$. 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 $[\![\alpha, \mu, \$]\!]_{c_i}^{\xi,\omega}$, where α represents a set of actors whose computation is managed by the cyberorg,; μ is a set of actor messages with local or remote recipients; \$ is the amount of eCash in the cyberorg;; ξ and ω 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.

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 $n-1$.

$$\begin{aligned}&\langle [\![\{\,[\,n \circ e\,]_a, \alpha\}, \mu, \$]\!]_{c_1}^{\xi,\omega}, \Gamma|\mathcal{M}|\mathcal{C}|t(c_1), \Theta|\mathcal{D}\rangle \\ &\quad \longrightarrow \langle [\![\{\,[\,(n-1) \circ e\,]_a, \alpha'\}, \mu', \$]\!]_{c_1}^{\xi',\omega'}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle \\ &T_{c_1}(st(c_1), co(c_1)) = a, n > 0\end{aligned}$$

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 α to α', and the multiset of messages from μ to μ'. 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 e'. 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.

$$\begin{aligned}&\langle [\![\{\,[\,0 \circ e\,]_a, \alpha\}, \mu, \$]\!]_{c_1}^{\xi,\omega}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle \\ &\quad \longrightarrow \langle [\![\{\,[\,n \circ e'\,]_a, \alpha'\}, \mu', \$]\!]_{c_1}^{\xi',\omega'}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle\end{aligned}$$

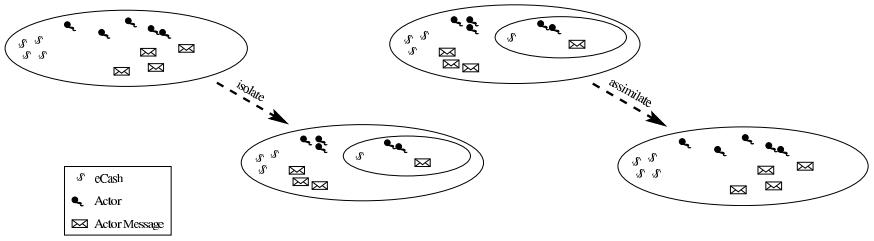
If there is a cyberorg c_2 being hosted by c_1 and the new tick is to be passed on to c_2, then, the tick is redirected to c_2, and an amount of eCash Δ representing the cost of the tick – determined by the contract $co(c_1, c_2)$ between c_1 and c_2, and $st(c_1)$, the state of c_1 – is transferred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

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

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

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

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.



a) CyberOrg Isolation b) CyberOrg Assimilation

Fig. 1. Creation and Absorption

Creation and Absorption. As illustrated in Figure 1a, 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, α_2, a subset of the messages, μ_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.

$$\langle [\![\{ \, [0 \circ \texttt{isolate}(\alpha_2, \mu_2, \$_2, \xi_2, \omega_2)\,]_\phi, \alpha_1 \}, \mu_1, \$_1]\!]_{c_1}^{\xi_1,\omega_1}, \Gamma | \mathcal{M} | \mathcal{C} | \Theta | \mathcal{D} \rangle$$
$$\longrightarrow \langle [\![\alpha_1 - \alpha_2, \mu_1 - \mu_2, \$_1 - \$_2]\!]_{c_1}^{\xi_1',\omega_1'}, [\![\alpha_2, \mu_2, \$_2]\!]_{c_2}^{\xi_2,\omega_2}, \Gamma | \mathcal{M}' | \mathcal{C}' | \Theta | \mathcal{D} \rangle$$
$$\alpha_2 \subset \alpha_1, \mu_2 \subset \mu_1, \$_2 \leq \$_1, c_2 \text{ fresh}$$

As shown in Figure 1b, 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.

$$\langle [\![\alpha_1, \mu_1, \$_1]\!]_{c_1}^{\xi_1,\omega_1}, [\![\{\texttt{[}0 \circ \texttt{asmlt]}_\phi, \alpha_2\}, \mu_2, \$_2]\!]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle$$
$$\longrightarrow \langle [\![\alpha_1 \cup \alpha_2, \mu_1 \cup \mu_2, \$_1 + \$_2]\!]_{c_1}^{\xi_1',\omega_1'}, \Gamma|\mathcal{M}'|\mathcal{C}'|\Theta|\mathcal{D}\rangle$$
if $< c_1, c_2 > \in \mathcal{M}$, $\nexists c_3$ such that $< c_2, c_3 > \in \mathcal{M}$

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:

$$\langle [\![\alpha, \mu, \$]\!]_{c_1}^{\xi,\omega}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle$$
$$\longrightarrow \langle [\![\{\texttt{[}n \circ \texttt{search}(\xi)\texttt{]}_\phi, \alpha\}, \mu, \$]\!]_{c_1}^{\xi,\omega}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle$$
if $\xi > av(co(c_1, c_2))$ 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 cyberorgs which may offer needed ticks for an acceptable price.

$$\langle [\![\{\texttt{[}0 \circ \texttt{search}(\xi_1)\texttt{]}_\phi, \alpha_1\}, \mu_1, \$_1]\!]_{c_1}^{\xi_1,\omega_1}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle$$
$$\longrightarrow \langle [\![\{\texttt{[}n \circ \texttt{negotiate}(C)\texttt{]}_\phi, \alpha_1\}, \mu_1, \$_1]\!]_{c_1}^{\xi_1,\omega_1}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle$$
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].

$$\langle [\![\{\texttt{[}0 \circ \texttt{negotiate}(C)\texttt{]}_\phi, \alpha_1\}, \mu_1, \$_1]\!]_{c_1}^{\xi_1,\omega_1}, [\![\alpha_2, \mu_2, \$_2]\!]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}|\mathcal{C}|\Theta|\mathcal{D}\rangle$$
$$\longrightarrow \langle [\![\{\texttt{[}n \circ \texttt{migrate}(c_2)\texttt{]}_\phi, \alpha_1\}, \mu_1, \$_1]\!]_{c_1}^{\xi_1,\omega_1}, [\![\alpha_2, \mu_2, \$_2]\!]_{c_2}^{\xi_2,\omega_2}, \Gamma|\mathcal{M}|\mathcal{C}'|\Theta|\mathcal{D}\rangle$$
if $C \neq \phi$, where $c_2 \in C = \{c_k, c_{k+1}, \cdots, c_l\}$ such that $\xi_1 < \omega_2, \mathcal{C}' \supseteq \mathcal{C}$

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

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

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:

$$\langle [\![\{ \texttt{[0} \circ \texttt{negotiate}(C)\texttt{]}_{\phi}, \alpha_1 \}, \mu_1, \$_1]\!]_{c_1}^{\xi_1,\omega_1}, \Gamma | \mathcal{M} | \mathcal{C} | \Theta | \mathcal{D} \rangle$$
$$\longrightarrow \langle [\![\alpha_1, \mu_1, \$_1]\!]_{c_1}^{\xi_1',\omega_1}, \Gamma | \mathcal{M} | \mathcal{C} | \Theta | \mathcal{D} \rangle$$
if $C = \phi \vee \nexists c_i \in C = \{c_k, c_{k+1}, \cdots, c_l\}$ such that $\xi_1 < \omega_i$

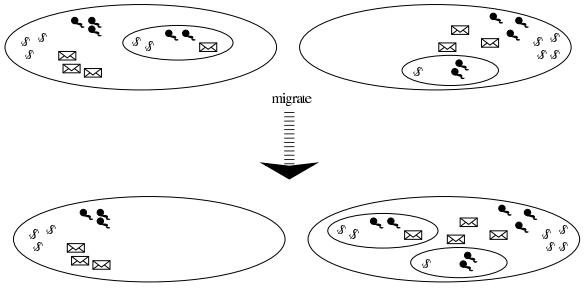


Fig. 2. Cyberorg Migration

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 2.

$$\langle [\![\{ \texttt{[0} \circ \texttt{migrate}(c_2)\texttt{]}_{\phi}, \alpha_1 \}, \mu_1, \$_1]\!]_{c_1}^{\xi_1,\omega_1}, [\![\alpha_2, \mu_2, \$_2]\!]_{c_2}^{\xi_2,\omega_2}, \Gamma | \mathcal{M} | \mathcal{C} | \Theta | \mathcal{D} \rangle$$
$$\longrightarrow \langle [\![\alpha_1, \mu_1, \$_1]\!]_{c_1}^{\xi_1',\omega_1'}, [\![\alpha_2, \mu_2, \$_2]\!]_{c_2}^{\xi_2',\omega_2'}, \Gamma | \mathcal{M}' | \mathcal{C} | \Theta | \mathcal{D}' \rangle$$
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:

$$\langle [\![\{ \texttt{[0} \circ \texttt{migrate}(c_2)\texttt{]}_{\phi}, \alpha_1, \mu_1, \$_1]\!]_{c_1}^{\xi_1,\omega_1}, [\![\alpha_2, \mu_2, \$_2]\!]_{c_2}^{\xi_2,\omega_2}, \Gamma | \mathcal{M} | \mathcal{C} | \Theta | \mathcal{D} \rangle$$
$$\longrightarrow \langle [\![\alpha_1, \mu_1, \$_1]\!]_{c_1}^{\xi_1',\omega_1'}, [\![\alpha_2, \mu_2, \$_2]\!]_{c_2}^{\xi_2,\omega_2}, \Gamma | \mathcal{M} | \mathcal{C} | \Theta | \mathcal{D} \rangle$$
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 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 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.

3 Prototype

A prototype implementation of CyberOrgs has been built using *Actor Foundry* [12], 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.

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.

3.1 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.

```
public void checkForTriggers() {
        if (activeActors.numElements() > 30)
                chores.enqueue("isolate");
        if (activeActors.numElements() < 3)
                chores.enqueue("assimilate");
        if (neededTicks > myTickRate)
                chores.enqueue("migrate");
}
```

Fig. 3. `checkForTriggers` method.

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 3 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 a particular primitive operation is to be carried out. For example, a possible implementation of the `initiateMigrationSequence` method for carrying out the migrate primitive is shown in Figure 4.

```
public void initiateMigrationSequence() {

    // ask YP 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.
}
```

Fig. 4. `initiateMigrateSequence` method

In addition to being triggered in response to the state, primitive cyberorg operations may also be explicitly requested by application actors. A cyberorg class may override default methods for servicing such requests.

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,[3] and any price for buying ticks that the cyberorg has enough *eCash* to pay.

Application Actors Implementation of a class of application actors subclasses from the `AppActor` class. The class defines methods describing behaviors for the application actors as they are defined in subclasses of the `Actor` class in *Actor Foundry*. 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`".

[3] meaning that the price paid for obtaining the ticks is lower than the price at which they are being sold.

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.

3.2 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.

4 Scheduling Cyberorgs

An important hurdle in efficiently implementing cyberorgs is the model's hierarchical structure. A naive way to enforce the hierarchical schedule of a cyberorg tree would be by implementing a hierarchy of schedulers. The overhead incurred by such a hierarchical scheduler would be prohibitive.

It turns out that enforcing cyberorgs' hierarchical distribution of cpu time does not require a hierarchical scheduler. Because availability of resources is in terms of what is available to the root cyberorg, and the availability for each cyberorg is in terms of resources its parent possesses (as stipulated by their contract), absolute availability of resources for each cyberorg can be maintained by simply looking at the contract and the parent's resources. By simple induction, the absolute resource availability for each cyberorg can hence be maintained in time proportional to the number of changes. Consequently, a global schedule can be created in which each cyberorg receives the resources it is promised as a function of the resources entering the system.. In other words, instead of launching a new scheduler for each cyberorg, all cyberorgs' internal schedules are composed into a single *flat schedule* of actors which is equivalent to the hierarchical schedule. Maintenance of the *flat schedule* can happen on the fly in response to primitive cyberorg operations, with a constant cost for each type of update.

A number of experiments were carried out on a prototypical Java implementation of the efficient scheduler for cyberorgs, to compare the overhead with the overhead of using a simple fair scheduler, or of letting Java's default scheduler schedule the threads.

4.1 CyberOrg Scheduler

The scheduler is implemented using two classes. The `Scheduler` class defines a thread scheduler which simply schedules threads (corresponding to actors at the tree's

leaves) for amounts of time for which they are to be scheduled. The scheduler uses Java's `suspend` and `resume` primitives to schedule threads. Another class, `ScheduleManager`, defines an update manager which receives requests for updating the cyberorg tree, and carries out the required changes in the flat schedule.

There are two parameters used by the system for managing the overhead by adjusting the granularity of control. Parameter `smallestSlice` puts a lower limit on how small a request for time slice can be, and parameter. However, requests for smaller time slices are not outrightly rejected. When a time slice lower than `smallestSlice` is requested, the time slices of each thread are scaled up so that the newest thread receives at least `smallestSlice`. The cost of this scale-up is in the total amount of time that one cycle of the scheduler takes, which coarsens the granularity of control. The second parameter, `largestSlice`, puts an upper limit on the size of a slice. This parameter becomes relevant at the time of accommodating a request for a time slice smaller than `smallestSlice`. If the scale-up required to award the new time slice is such that the highest time slices becomes larger than `largestSlice`, then the request is denied.

Threads	CyberOrg Scheduler			Fair Scheduler						No
(Actors)	Height	Cyberorgs	Time	Max	Time	Mean	Time	Min	Time	Scheduler
10	2	4	356	14	272	8	280	2	334	319
50	4	17	1040	183	999	33	1087	2	1022	1020
100	3	15	1967	146	1878	17	1969	2	2016	2201
200	4	27	4058	250	3720	15	3750	2	3775	3399
300	5	40	5372	370	5412	19	5908	2	6074	5017
400	5	67	7544	356	6685	21	7202	2	7931	6299
500	5	59	8946	239	7823	13	8313	2	9043	7922
600	5	71	11040	352	10121	11	10507	2	10943	9938
700	5	102	13866	390	11607	11	12736	2	13291	10906
800	5	74	14754	330	14203	11	14359	2	15614	12892
900	6	129	17061	634	15617	16	16177	2	16568	13998
1000	6	140	18736	324	16548	14	17715	2	18087	16781

Table 1. Comparison of different scheduling choices for cpu intensive computations. Time is in milliseconds. Height is the height of the cyberorg tree; Cyberorgs is the final number of cyberorgs in the system. Columns min, mean and max show the time slices used by a fair scheduler corresponding to the minimum, mean and maximum of time slices awarded by the cyberorg scheduler for the same number of threads.

4.2 Experiments

Experiments were carried out for comparing performance of four broad scheduling choices for cpu intensive computations (Table 1). The first choice was to allow Java's default thread scheduler to schedule the concurrently executing threads carrying out the computations; second was a fair scheduler that awarded uniform time slices to all

threads; third was the cyberorg scheduler for scheduling cyberorgs according to the requested time slice allocations. The final choice was to sequentialize all computations carried out concurrently in the previous cases, to be carried out by a single thread.

Because there is a relationship between the sizes of time slices and the scheduling overhead, to keep the comparison fair, the fair scheduler experiments were carried out with three different time slices, corresponding to the smallest, mean and largest time slices for which the cyberorg scheduler scheduled its actor threads.

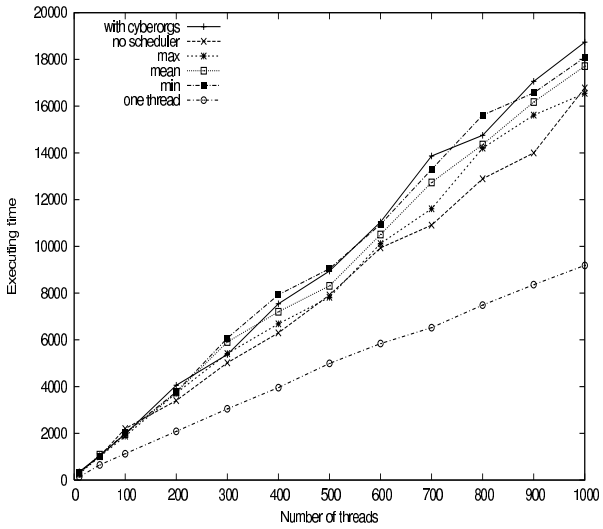


Fig. 5. Scheduler Performance

As illustrated by the graph corresponding to Table 1 in Figure 5, no significant overhead is incurred in enforcing the hierarchical schedule of a tree of cyberorgs. Specifically, the overhead is proportional to the number of threads (actors) in the system irrespective of the number of cyberorgs and the height of the cyberorg tree.

5 Conclusions

Agents sharing an execution environment invariably compete for available resources, possibly in ways impacting global performance of a multi-agent application. However, resource usage is one aspect of multi-agent behavior which naturally lends itself to abstraction. CyberOrgs offer a model for hierarchical coordination of resource usage by multi-agent applications in a network of peer-owned resources, allowing multi-agent applications to execute in an environment of predictable resource availability. The model achieves a separation of concerns by representing resource requirements of an application separately from its functionality. We have introduced programming

constructs for implementing systems of cyberorgs as well as described scheduling techniques for efficient distribution of processor resource.

Work is ongoing to develop a distributed version of the cyberorg scheduler to support scheduling of cyberorgs spanning a network of computers. Additionally, the method of flattening a hierarchical schedule is being generalized to achieve efficient distribution of other computational resources, such as network bandwidth.

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. A. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufman Publishers, San Mateo, California, 1988.
5. L. Gasser. DAI approaches to coordination. In N. M. Avouris and L. Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*, pages 31–51. Kluwer Academic Publishers, 1992.
6. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
7. C. Hewitt and P. de Jong. Open systems. In J. Mylopoulos, J. W. Schmidt, and M. L. Brodie, editors, *On Conceptual Modeling*, chapter 6, pages 147–164. Springer Verlag, 1984.
8. N. Jamali. *CyberOrgs: A Model for Resource Bounded Complex Agents*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.
9. N. R. Jennings. Commitments and conventions: The foundation of coordination in multiagent systems. *The Knowledge Engineering Review*, 8(3):223–250, 1993.
10. W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, 1995.
11. W. A. Kornfeld and C. Hewitt. The scientific community metaphor. *IEEE Transactions on System, Man, and Cybernetics*, 11(1):24–33, January 1981.
12. Open Systems Laboratory. The Actor Foundry: A Java-based actor programming environment. *Available for download at ⟨http://www-osl.cs.uiuc.edu/foundry⟩*.
13. C. Manning. Introduction to programming actors in acore. In C. Hewitt and G. Agha, editors, *Towards Open Information Systems Science*, chapter 2, pages 33–80. MIT Press, Cambridge Mass, 1990.
14. L. Moreau and C. 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.
15. L. Moreau and C. Queinnec. Distributed and Multi-Type Resource Management. In *ECOOP'02 Workshop on Resource Management for Safe Languages*, Malaga, Spain, June 2002.
16. J. E. White. Telescript Technology: The Foundation for the Electronic Marketplace. Technical report, General Magic Inc., Mountainview, CA, 1994.