A Scalable Approach to Multi-Agent Resource Acquisition and Control

Nadeem Jamali
Department of Computer Science
University of Saskatchewan
176 Thorvaldson Bldg., 110 Science Place
Saskatoon, SK, Canada, S7N 5C9
n.jamali@usask.ca

Xinghui Zhao
Department of Computer Science
University of Saskatchewan
176 Thorvaldson Bldg., 110 Science Place
Saskatoon, SK, Canada, S7N 5C9
x.zhao@usask.ca

ABSTRACT

Scalable coordination is a key challenge in deployment of 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. A prototype implementation of CyberOrgs is described and expressive power of the programming constructs is illustrated with examples.

Hierarchical control presents challenges in scalability. However, CyberOrgs make some types of resource coordination more 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. Experimental results are presented which demonstrate scalability of this approach.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Performance, Design, Economics, Experimentation, Languages

Keywords

CyberOrgs, Actors, Hierarchical Control, Resource Control, Coordination, Control Reification

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 [11].

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 purges misbehaving agents, so long as the mechanism relies solely on the observation of individuals' suspicious activity, by the time the system reacts, it may be too late: other agents have potentially been created about whose behavior the system will know nothing until it has observed them individually.

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.

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 c^{th} cycle of the scheduler, the number agents is $(n(1 - p/k))^c$, which represents an exponential growth.

An alternate approach to control is by bounding resource consumption at the outset, and limiting resources available to a com-

AAMAS'05, July 25-29, 2005, Utrecht, Netherlands.

putation 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 [13] 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* [14]. *Sponsor actors* accompanied computation requests, and they carried ticks that could be used in processing a request. Using a similar, in *Telescript* [17], processes were awarded funds in terms of *teleclicks* which they were supposed to use to accomplish their results.

The Quantum [15] framework is the most relevant to our work. Motivated by the need for managing finite resources shared by multiple computations, Quantum models resources as *energy* which computations require for execution. Computation 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 [16] to handle management of distributed and multi-type resources, which does address migration in a limited manner. Our approach goes a step further by explicitly modeling resource trade separately from management of owned resources. This separation is required for resource acquisition in the setting of peer-owned resources; it also allows dynamic pricing to be modeled. Additionally, our approach to representing resources and their propagation allows modeling of resource expiration. This is a more accurate representation of computational resources which if unused at an instant, that instant's use of the resource can no longer be reclaimed.

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 avail themselves 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 distributed computations (to be referred to as computations contained in the cyberorg), and an amount of *eCash* in a shared currency. Cyberorgs act as principals in a market of distributed resources, where they may use their *eCash* to buy or sell resources among themselves. A cyberorg may use the resources so acquired for carrying out its computations, or it may sell them to other cyberorgs.

CyberOrgs treat computational resources as being defined in time and space. In other words, a resource is not available for use before or after the instant of time at which it exists. 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. Cyberorgs may pre-pay to buy resources which will exist in the future.[1] Cyberorg b may use the resources it owns only if the resources exist at a time when the cyberorg is being hosted by a. In other words, after signing a contract, a cyberorg must migrate to the prospective host cyberorg in order to avail itself of newly acquired resources. Additionally, if b migrates from a while it owns future resources through a contract with a, it cannot use those resources except if it eventually returns to a and if it possesses resources which have not yet expired.

[1]Recall that resources only exist at instants of time.

Example: Distributed Weather Forecast

Consider a distributed weather forecasting system for analyzing weather data and generating alerts and warnings about specific threats. Computational resources available to such a system may be focused on particular weather systems or on population centers. As a threatening weather system moves from one city to another, dedicated resources available for tracking the weather system may need to become available to the cities requiring them for assessing the impact on them.

A CyberOrgs implementation of such a system could geographically organize weather forecasting computations and the resources available to them. In other words, *eCash* may be made available to a hierarchy of geographic regions to acquire resources to carry out computations necessary for assessing weather related threats. A cyberorg would be in charge of each region; it will use the *eCash* dedicated to monitoring the region to acquire resources for supporting the forecasting computations. Simultaneously, other cyberorgs with *eCash* for tracking specific active weather systems would follow the weather systems, migrating from one regional cyberorg to another, releasing *eCash* required by the current host cyberorg.

As a threatening weather system enters a region, the region's cyberorg may decide to create specialized cyberorgs for larger population centers, allocating resources to each in the form of *eCash* based on the threat level and complexity of data to be analyzed (which may include weather data as well as information about the local sub-region). As the weather system moves on, sub-regional cyberorgs may *assimilate* into their host cyberorgs, returning the *eCash*.

2.1 Formalizing CyberOrgs

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 primitive agents called *actors*.

2.1.1 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, call-by-value λ-calculus is extended in [3].

Primitive agents are naturally modeled by the Actor formalism. In fact, Actors is the *de facto* model underlying many implementations of mobile agent systems. Actors are autonomous and persistent, and they are inherently concurrent and autonomous, enabling efficient parallel execution [12] and facilitating mobility [2].

2.1.2 Resources

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 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 basis for measurement of potential progress; 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 represented explicitly.

2.1.3 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, or 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 proportion of the total number of ticks inserted into the system at the root cyberorg. Contracts also determine the costs which the clients are supposed to be charged for the ticks they receive. A contract may allow a client cyberorg to pre-pay for ticks that exist in the future.

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

If there is a cyberorg b being hosted by a and the new tick is to be passed on to b, then, the tick is redirected to b, and an amount of *eCash* representing the cost of the tick – determined by the contract between a and b, and the state of a – is transferred from b to a. If a cyberorg runs out of *eCash*, and the resources it requires are not available free of cost, its actors become dormant until it receives *eCash* from another cyberorg. For instance, the cyberorg may own ticks defined in the future, which it may give to hosted cyberorgs, and receive *eCash* in return. Alternatively, a hosted cyberorg may assimilate, releasing its *eCash*, as shown in the next section.

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

2.1.4 CyberOrgs Primitives

In addition to transitions corresponding to progress in actor computations, there are a number of transitions in the system which correspond to CyberOrgs primitives. These transitions happen through invocations of CyberOrgs commands from *helper* actors, which in turn are created by the cyberorg's *facilitator* actors. A facilitator actor monitors the state of the current host as well as the cyberorg's resource requirements, and creates helpers to carry out CyberOrgs 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 1(a), 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 which is hosted in locally according to a contract imposed by the creating cyberorg.

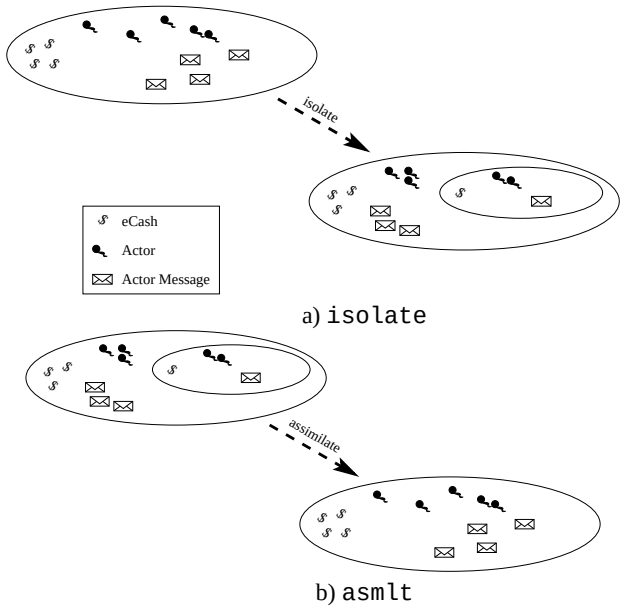

Figure 1: Creation and Absorption

As shown in Figure 1(b), 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.

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 the resource requirements of the cyberorg's functional or non-functional (e.g. spatial, temporal) behavior exceed what is available by its contract with the current host. 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:[2]

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.

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 want-

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

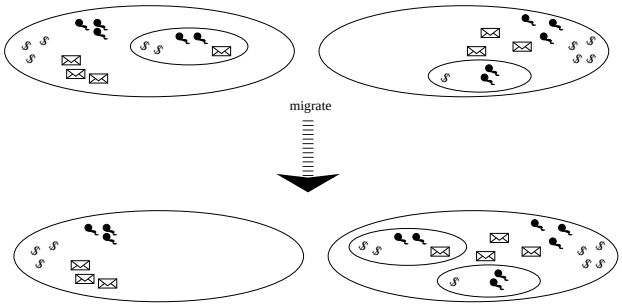

Figure 2: Cyberorg Migration

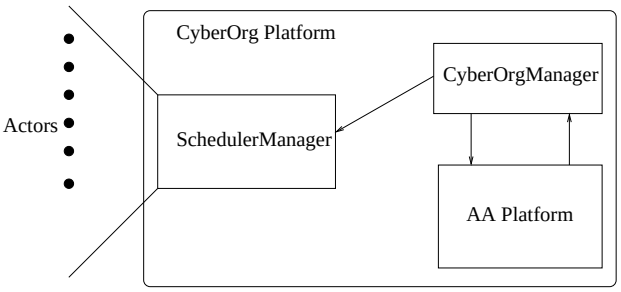

Figure 3: CyberOrg Platform Structure

ing 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.

If there are no cyberorgs which can serve cyberorg a's resource requirements, no negotiation can happen, and a adapts to its current resource availability.

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.

   If a contract is not successfully negotiated, the cyberorg adapts to its current resource availability.

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. 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.

Distribution of resources among a cyberorg's actors is a local decision, and no specific strategy is mandated by the model. As a result, the local resource distribution strategy of a cyberorg can be used for fine-grained control of how a cyberorg's actors consume resources, to achieve functional and non-functional goals of the application. When an actor is not provided ticks, it becomes dormant.

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.

A more formal treatment of the operational semantics may be found in [9].

3. IMPLEMENTATION OF CYBERORGS

We have implemented CyberOrgs by extending *Actor Architecture* [10] – a Java library and run-time system for supporting actors. In this implementation, every actor requires processor time resource to carry out its computation, and the resource is received by the actor from the cyberorg containing it.

Actor Architecture (AA) is a middleware for providing an execution environment for actors. An instance of the AA run-time system is called a platform, which supports actors executing on one physical machine. AA has several layers of components that provide different services: *Actor Management Service* keeps and manages state information for all actors in the platform including both running and mobile actors; *Message Delivery Service* handles all messages in the current AA platform; *Message Transport Service* passes messages between AA platforms; *Advance Service* provides match-making and brokering services.

In Actor Architecture, actors may run concurrently and exist across distributed systems; however, resource consumption is not accounted for. Specifically, the Java VM is left with the task of scheduling actor threads.

Our implementation of CyberOrgs adds two key components to a AA platform: CyberOrg Manager and Scheduler Manager, as illustrated in Figure 3.

CyberOrg Manager

CyberOrg Manager adds run-time support for a system of cyberorgs. Every actor is contained in a cyberorg, and the actors receive the processor resource they need from the cyberorgs containing them.

Each cyberorg holds a number of actors representing a concurrent computation, an amount of *eCash* to support those actors, processor resource defined in time and space (represented by the cyberorg's contract with its host) available for carrying out the computation; it also hosts other cyberorgs. A cyberorg has its own strategy to allocate available resources among its actors and hosted cyberorgs. This strategy is implemented in the form of a *facilitator* actor, which is a special actor that serves as the active part of a cyberorg. Among other tasks, a facilitator triggers primitive CyberOrgs operations which react to changes in the environment as well as the the cyberorg's requirements.

Scheduler Manager

Scheduler Manager is made up of two parts: the cyberorg scheduler and the thread scheduler. The cyberorg scheduler keeps track of the hierarchical structure of the CyberOrgs hierarchy for a single platform and converts the hierarchical schedule represented by the structure into a flat schedule for the actor threads. This schedule, which contains times for which each actor is to be scheduled, is dynamically updated with changes in the hierarchical schedule as a result of invocation of CyberOrgs primitives and changes in the cyberorgs' local resource distribution.

The thread scheduler schedules actor threads contained in a queue for the amounts of processor time they are supposed to be scheduled. It uses Java's suspend and resume primitives[3] to schedule

[3]Although these primitives are deprecated, they are safe to use on actors because multiple threads do not access the same object.

actors in the queue. The time slices for actors are computed and updated dynamically by the cyberorg scheduler.

3.1 Programming Constructs

The API for CyberOrgs includes support for creating cyberorgs and actors as well as CyberOrgs primitive operations.

Creation

There are two types of creation in the CyberOrgs system: cyberorg creation and actor creation. Cyberorg creation method is called only once by the GUI or user program to create the first cyberorg in the system; future cyberorg creations happen as a result of invocation of the isolate primitive. Actor creation method is called by existing actors to create other actors, or by the cyberorg constructor in order to create the facilitator actor.

- Cyberorg creation:
  `CyberOrg createCyberOrg(long ticks, long ticksrate, long eCash, String facilitatorClass, Object[] args)`

where `ticksrate` is the rate of processor time that the new cyberorg would receive with respect to the cyberorg in one scheduling cycle; `eCash` is the number of eCash units that are used for buying processor resource from host cyberorg; `facilitatorClass` and `ags` identify the facilitator actor class for the cyberorg and the arguments for creating such a facilitator actor.

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

where `creator` is the unique name of the creator; `actorClass` and `ags` specify the class of the actor being created and the arguments used in the actor constructor. This method is used by one actor to create another actor.

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

where `host` identifies the creating cyberorg; `facilitatorClass` identifies the actor class of the facilitator and `args` specify the arguments to be used in constructing the facilitator actor. This is used by the cyberorg constructor at the time of creation of a cyberorg to create a facilitator actor.

CyberOrgs Primitives

Primitive CyberOrgs operations are called by the facilitator actor of the cyberorg.

- Isolation:
  `CyberOrg isolate(long eCash, ActorName[] actors, Contract newContract)`

where `eCash` is the amount of *eCash* that is given to the newly created cyberorg; `actors` is an array of existing actors that will be isolated into the new cyberorg; `newContract` is the contract imposed on the new cyberorg and its host cyberorg which specifies the ticks and ticks rate that the new cyberorg receives, as well as the cost of the resources in terms of *eCash* payments to be made.

- Assimilation: `CyberOrg assimilate()`

This primitive will cause assimilation of the cyberorg into its host.

- Migration:
  `void migrate(ActorName facActorOfDestCyberOrg, Contract newContract)`

where `facActorofDestCyberOrg` is the name of facilitator actor in the destination cyberorg which serves as the cyberorg's name; `newContract` is the negotiated contract between the migrating cyberorg and the intended host.

Negotiation

Before migration, a cyberorg needs to negotiate a contract with a prospective host. The negotiation is initiated by the facilitator actor of the cyberorg interested in migrating by a call of the negotiation primitive.

- Negotiation:
  `Contract negotiate(ActorName destFacilitatorActor)`

where `destFacilitatorActor` is the facilitator actor of the prospective future host cyberorg.

Example: Distributed Weather Forecast

We return to the motivating example of a distributed weather forecasting to illustrate the expressive power of programming constructs based on the CyberOrgs model. A programmer can implement application part of the code using the Actor constructs offered by Actor Architecture. Additionally, the resource management part of the system can be implemented separately by sub-classing from the `CyberOrg` and `FacilitatorActor` classes.

The system may have two types of cyberorgs: region and weather system specific. A regional cyberorg would manage resources (represented as *eCash*) for developing forecasting information for a region; a weather system cyberorg would manage resources dedicated to understanding a weather system of particular interest. In the event of localized weather activity, regional cyberorgs would isolate parts of their computations dedicated to effected sub-regions to form cyberorgs with independent control. At the conclusion of high local activity, cyberorgs for smaller regions may assimilate into cyberorgs for larger regions, relinquishing independent control of resources.

A weather system cyberorg would migrate from one regional cyberorg to another, taking with it dedicated resources to be made available to regions facing the system. Specifically, on arriving in a regional cyberorg, the weather system cyberorg would isolate part of its *eCash* into a new cyberorg, which migrates out to the regional cyberorg for assimilation. Figure 4 illustrates how this would be implemented as two methods of the weather system cyberorg's facilitator. Method `actionOnArrival` is invoked on the weather system cyberorg's arrival into a regional cyberorg. It checks if the region is still under threat; if it is, need for funds is assessed, the funds are isolated into a new `FundsCyberOrg` cyberorg, which is then asked to negotiate terms to migrate to the host region. Once a contract is negotiated and migration has happened, the `FundsCyberOrg` assimilates into the region, releasing its funds.

Figure 5 illustrates the delegation of sub-regional control of resources by a regional cyberorg. The `checkStatus` method of the regional cyberorg's facilitator checks whether resources have to be set aside for delegated control for a sub-region. If so, the smaller region's actors are isolated along with the necessary amount of *eCash*. Once local control is no longer required (for example, because a weather system has passed the sub-region), a regional cyberorg simply assimilates into the cyberorg of the enclosing region.

```
public void actionOnArrival() {
  if (w_sys.effects(reg) {
    long toOffer= Needs(w_sys, reg);
    CyberOrg FundsCyberOrg=
      isolate(toOffer, new ActorName[0],
        defChildContract);
  send(fundsCybFacil,
  "triggerFundMigration", hostCybFacil);
  }
}
public void triggerFundMigration
  (ActorName destination) {
  Contract offerSupport=
    negotiateWith(destination);
  if (offerSupport != null) {
    migrate(destination, offerSupport);
  }
}
```

Figure 4: Facilitator actor methods of WeatherSys CyberOrg. fundsCybFacil and hostCybFacil are facilitator actors of FundsCyberOrg and hostCyberOrg respectively

Most interestingly, if the resources available through the contract with the current host cyberorg are not sufficient, a regional cyberorg may negotiate a better contract with another cyberorg. In other words, although the cyberorgs for sub-regions are created by cyberorgs for larger regions, the sub-regional cyberorgs are free to migrate elsewhere in search of a better matching execution environment.

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

Figure 5: Method `checkStatus` in facilitator actor of regional cyberorg

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 even for a single processor.

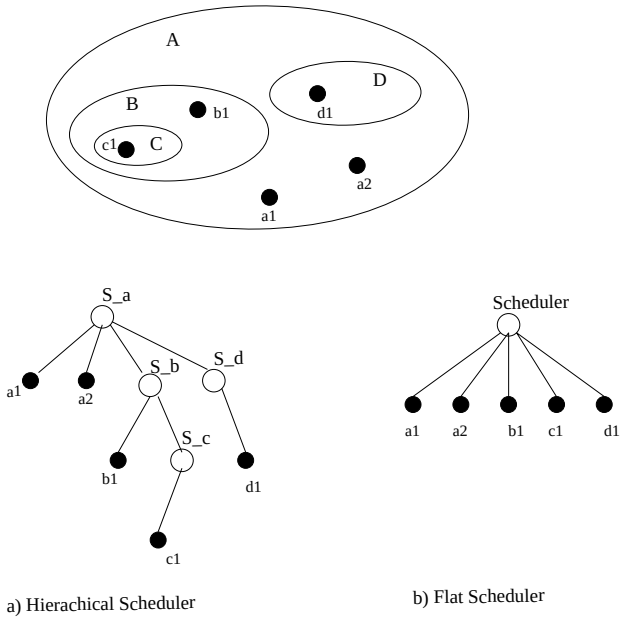

Figure 6: Scheduler Schemes for CyberOrgs: Flattening of the hierarchical schedule eliminates need for layers of schedulers

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 (Figure 6). 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 CyberOrgs 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, `Sche-duleManager`, 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 `smallest-Slice`. 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 `smallest-Slice`. 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.

Actors	CyberOrgs Scheduler			Fair Scheduler		
(Thrds)	Height	Cybs	Time	Max	Mean	Min
10	2	4	356	272	280	334
50	4	17	1040	999	1087	1022
100	3	15	1967	1878	1969	2016
200	4	27	4058	3720	3750	3775
300	5	40	5372	5412	5908	6074
400	5	67	7544	6685	7202	7931
500	5	59	8946	7823	8313	9043
600	5	71	11040	10121	10507	10943
700	5	102	13866	11607	12736	13291
800	5	74	14754	14203	14359	15614
900	6	129	17061	15617	16177	16568
1000	6	140	18736	16548	17715	18087

Table 1: Comparison of scheduling choices for cpu intensive computations. Time is in milliseconds. Height is final height of cyberorg tree; Cybs is the final number of cyberorgs. Columns min, mean and max show 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. The first choice was the cyberorg scheduler for scheduling cyberorgs according to the hierarchical schedule; second was a fair scheduler that awarded uniform time slices to all threads; third was to allow Java's default thread scheduler to schedule the concurrently executing threads carrying out the computations; 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.

All actor threads in the experiments carried out identical cpu-intensive computations. For the cyberorgs case, new cyberorgs were randomly isolated by existing cyberorgs during the course of the computation.

Table 1 shows typical results for the cyberorg scheduler and the fair schedulers with the three different time slice values. The first column shows the total number of actor threads created during the computation, which also determines the total amount of computation carried out. The next set of three columns shows (for the number of actors in each row) the final height of the cyberorg tree and the final number of cyberorgs in the system at the end of the computation, and the execution time for each case. The set of three columns that follows shows the execution times for the three cases of fair scheduler executions involving the number of actors in the row.

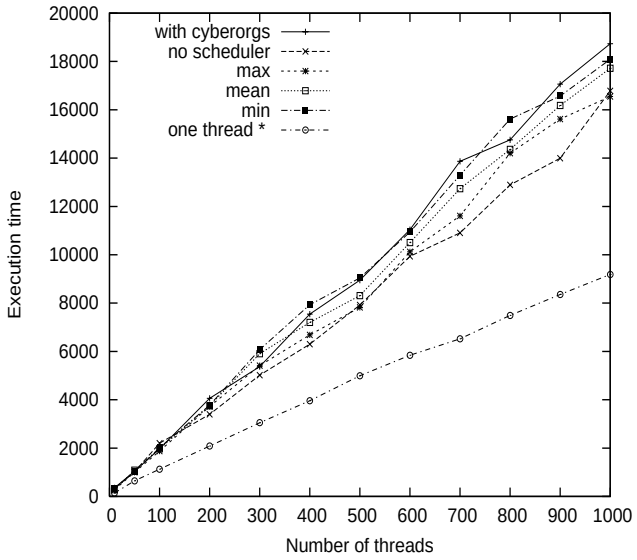

Figure 7: Scheduler Performance (*x axis for *one thread* case represents the computational load equivalent to the load of the number of threads). Cyberorgs do not incur significant amount of overhead beyond that incurred by techniques that do not provide resource control.

The graph in Figure 7 illustrates the results shown in Table 1, as well as results for the cases when the Java runtime system is allowed to schedule the threads, and when the entire work load is carried out by a single thread (to serve as a reference point). As the graph shows, no significant overhead is incurred in enforcing the hierarchical schedule of a tree of cyberorgs. Specifically, the overhead of enforcing CyberOrgs' hierarchical control is proportional to the number of threads (actors) in the system irrespective of the number of cyberorgs and the height of the cyberorg tree.

These experiments were carried out on a single processor with a monotonically increasing number of cyberorgs and without any migrations or assimilations. Although enforcing hierarchical control and isolating new cyberorgs (at a certain rate) has been shown to be inexpensive, changes in the hierarchy as a result of typical patterns of primitive CyberOrgs operations will require updating of the flat schedule in multiple ways, in line with the frequency of the primitive operations. Cyberorgs changing how they distribute their processor time among their actors too will require changes to be made in the flat schedule. Because contracts with client cyberorgs (and those involving cyberorgs further down in the tree) remain unchanged when a cyberorg migrates to another host, the cost of updates resulting from each migration is proportional the number of actors managed by the migrating cyberorg. Similarly, because a cyberorg may not assimilate if it is hosting other cyberorgs, the cost of each assimilate is proportional to the number of actors in the assimilating cyberorg. Finally, the cost of a cyberorg changing its distribution of processor resource among its actors is proportional to the number of actors being effected. Work is ongoing to study

overheads resulting from interesting patterns of CyberOrgs primitive operations, and to obtain analytical results relating overheads to the number and types of operations per unit of computation.

4.3 Distributed Hierarchical Scheduler

A distributed scheduler would control processor resources on a number of connected machines. However, implementing such a scheduler is complicated by communication delays. Specifically, a fine-grained cpu scheduler must be local to the processes it is scheduling. A distributed scheduler, therefore, must be a network of communicating local schedulers. Schedules for such a scheduler must explicitly address communication delays. In the context of CyberOrgs, it is possible to create such schedules by examining communication delays which can be estimated once the network resource is controlled. In other words, once an amount of network bandwidth has been secured for a cyberorg, cpu scheduling of its distributed actors may rely on predictable communication delays. Work is currently ongoing to implement efficient hierarchical control of network bandwidth and to develop a distributed version of the CyberOrgs scheduler.

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 an efficient prototype implementation and programming constructs for implementing systems of cyberorgs, and described scheduling techniques for efficient distribution of processor resource.

Preliminary experimental results show that CyberOrgs' hierarchical control does not incur significant overhead. Relationship between overhead and interesting patterns of CyberOrgs primitive operations is being studied. Work is ongoing to generalize the method of flattening a hierarchical schedule to achieve efficient distribution and control of other computational resources. A distributed version of the cyberorg scheduler is under development which relies on predictability of communication delays resulting from effective control of network bandwidth.

Acknowledgements

This research is supported in part by an NSERC Discovery Grant.

6. 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. Jamali and X. Zhao. Hierarchical resource usage coordination for large-scale multi-agent systems. In T. Ishida, L. Gasser, and H. Nakashima, editors, *LNAI: Massively Multi-agent Systems I*, volume 3446, pages 40–54. Springer Verlag, 2005.

[10] M. Jang and G. Agha. On efficient communication and service agent discovery in multi-agent systems. In *Proceedings of the Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS '04)*, pages 27–33, Edinburgh, Scotland, May 2004.

[11] N. R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review*, 8(3):223–250, 1993.

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

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

[14] 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.

[15] 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.

[16] 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.

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