Version in *IEEE Intelligent Systems*, Special Issue on Agents, March/April 1999

An Actor-based Architecture for Customizing and Controlling Agent Ensembles

Nadeem Jamali, Prasannaa Thati and Gul A. Agha
Department of Computer Science
1304 W. Springfield Avenue
University of Illinois
Urbana, IL 61801, USA
Email: {jamali | thati | agha}@cs.uiuc.edu

Consider a distributed internet-based market place with sellers and buyers represented by autonomous mobile agents. Individual users of the system may create agents to purchase or offer goods or services; these agents may travel over the network searching for bargains or for potential markets. What framework would best support such a system?

An opportunistic user would always selfishly seek the best deal, which may translate into having access to as much information as possible (in a managable form, of course). In a distributed environment, this would mean spawning a very large number of agents, possibly organized as a tree, to disperse over the network. In a more volatile market needing local decisions, seller and buyer agents would want to be omnipresent. In both cases, agents serving the same interests will often need some type of coordination; an ability to coordinate the behavior of agents in agent ensembles is a key challenge for Distributed AI.

From the perspective of a node hosting such activity, there has to be some incentive to allow it. Malicious or erroneous agents may threaten the node not only by attempting to access specific resources, but also by the degree to which they use them. More importantly, chaotic behavior may emerge from otherwise reasonable behaviors of agents in a large ensemble. In recent past there have been examples where outcomes of collections of autonomous processes have resulted in this kind of phenomena. For instance, a Haitian ferry sank last September as the passengers rushed to one side to disembark. A 1996 power outage in Oregon caused a cascading outage along the entire US West Coast because of lack of cooperation among utilities. In our distributed market example too, seller and buyer agents may be driven in large numbers towards a new market or a discount sale, possibly causing node and network failures along the way.

A platform for supporting multi-agent ensembles needs to provide scalable mechanisms for safe and efficient execution over open networks of computers. The underlying platform of an agent system must control ways in which resources are accessed and managed by representing resource allocation policies at the agent level, borrowing ideas from previous work in subject areas as diverse as operating systems and economics.

1 Defining Agents

A common type of commercially available agents is the personal assistants that perform a large number of light weight queries in search of some information. Personal assistants perform functions such as finding the best travel fares, monitoring product or stock prices, or searching academic articles related to a certain area of research. These agents may even have the decision making authority to make binding contracts on behalf of a user, such as by purchasing something using a credit card number. Other types of agents use a variety of filtering mechanisms to make the huge amount of information available over (say) the Internet more manageable for human consumption. These agents can be seen as examples of (stationary) personal agents.

In contrast, in an open system, agents may migrate from one node to another searching for computation environments suitable for completing their tasks at affordable costs. These agents may also spawn child agents to pursue subtasks. This makes it important to study ways of controlling the *resources* that such agents or their ensembles could use in serving some particular interest. On the one hand, we need a *bounded resources* model to control the amount of computational resources consumed by agents serving an interest; on the other, we need a *bounded autonomy* model for allowing coordination among agents.

Agents may be represented as actors (Figure 1). Actors are self-contained, interactive, autonomous components of a computing system that communicate by asynchronous message passing [1, 2]. New actors may be created and mail addresses of actors communicated in messages. We extend the Actor model to explicitly model the location of agents on particular *hosts* and the *bounded computational resources* that they may use. Hosts are actors that manage physical and logical resources of a node and offer them to agents interested in paying for them. A *universal currency* is used to pay for the cost of these resources. The behavior of an agent may be interpreted in a suitable framework, e.g., the belief, desire, intent model [6]. Agents are persistent, have relatively long-lived goals describing the functional aspect of what they are doing, and have computational engines which serve as mechanisms for achieving these goals. In addition to the functional component, these computational engines include a resource utilization strategy. Of course, all these aspects of an agent may evolve dynamically.

Using the basic actor primitives for creating a new agent, sending an asyn-

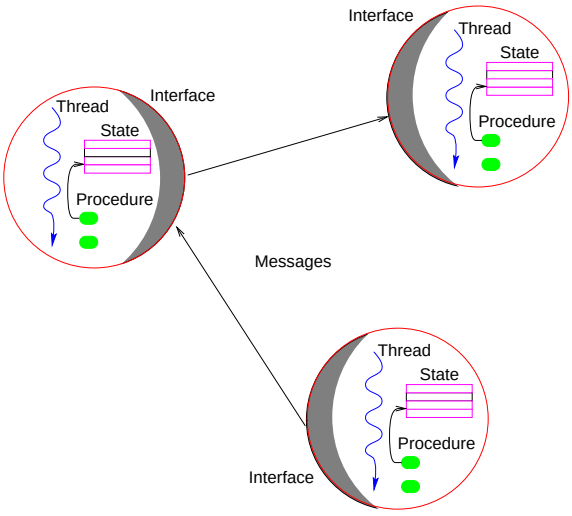

Figure 1: Actors encapsulate a thread and state. The interface is comprised of public methods which operate on the state.

chronous message to another agent, and changing the agent's own behavior, distributed systems can be dynamically configured. New agents can be created and connections between agents can be made and broken as computation proceeds. Thus the Actor model does not require that the structure or shape of a computational problem be completely determined, or that the execution resources be fixed, before work on solving it can be initiated.

This model abstracts over issues of low-level synchronization by encapsulating the state of an object and its execution thread, and limiting communication to asynchronous message passing.

1.1 Mobility

Consider a knowledge acquisition problem where we want to build a semantic network representing the information available in a large knowledge space, distributed over a set of independent sites. (Figure 2).

To initiate searches in the subspaces, a new agent can be created at each site and assigned the task of searching locally. Each of these agents can develop a semantic network for the information available locally (possibly creating its own child agents and and assign them local subspaces), and transmits it to the

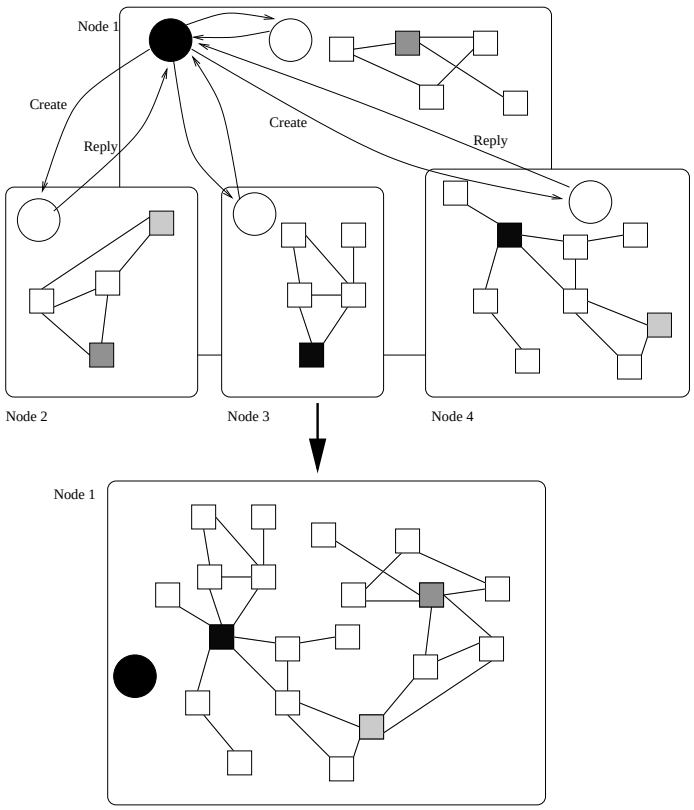

Figure 2: Knowledge acquisition by agents in a distributed space

home site. The home site finally merges the networks collected from all other sites to produce a final result.

An agent in an open system may want to initiate computation at a different location for a variety of reasons. The new location may provide a more suitable computational environment, it may offer cheaper resources, or it may have data needed by the agent for completing its task.

Although creating agents remotely suffices for the purpose of our example, consider a variation where a single agent must travel to different nodes, building the semantic network incrementally, and relying on the network built thus far to decide where to go next. Mobility seems a more natural abstraction to address this problem.

True migration as depicted in Figure 3, involves capturing the current com-

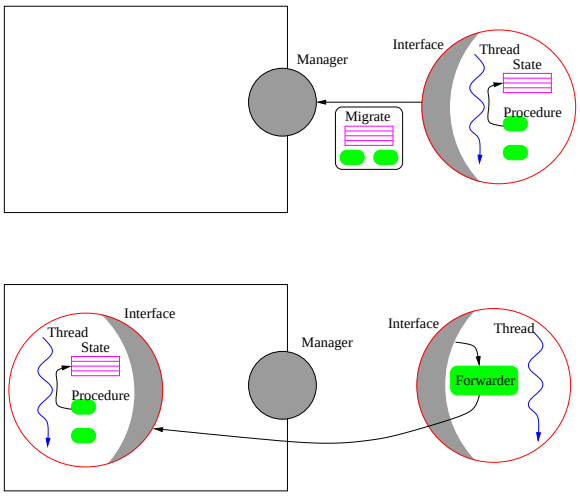

Figure 3: Migration of an agent. New location receives the agent's state and procedures to create a new agent locally

putational state of an agent and shipping it over to the remote node's manager, along with the procedures describing the agent's behavior. The remote manager creates a new actor with the behavior and the received state. Similar to true migration is a language facility for migrating an agent when it is inactive. This is semantically equivalent to remote creation of an agent given a behavior, but the two cannot always be interchanged. For example, remote creation may be cheaper when the amount of information to describe the new agent is less than the size of the agent itself.

Another important distinction between remote creation and migration is that the identity of an agent remains unchanged after migration, whereas remote creation results in creation of a new name. Because of this, an agent requesting a remote creation would have to wait for the identity of the new agent before communicating with it. Alternatively, an agent may create a new actor locally and request to migrate it.

A common abstraction for allowing migration in existing agent system implementations is in the form of possibly dynamic itineraries, which are lists of node/method pairs, describing the sequence in which the agent must travel to different nodes, and the methods it must execute at those hosts. This scheme

too does not allow migration in the middle of a method's execution.

Language constructs for adaptation are also very useful for supporting migration. In networks of heterogeneous systems, agent may have to migrate to nodes with different architectures, and a static specification of the behavior may not always be compatible with the new host. This problem can be addressed by allowing agent behaviors to analyze and modify themselves as needed.

1.2 Controlling Resources

To support a system where agents can use resources available "elsewhere" in a satisfactory way, it is important to have some notion of an economy. Such an economy would provide the basis on which nodes would allow agents to use their resources, and would serve as an environment that would enable nodes and agents to get into binding contracts about the services needed.

Resource allocation in multi-agent systems is a problem that raises issues of reciprocity as well as performance and security concerns. Nodes on the world-wide web, for instance, may be willing to be part of a multi-agent system if they receive something in return for allowing foreign agents to use their resources. From the performance and security perspective, agents migrating to a node may exhibit undesirable resource consumptive behaviors, either individually, or as ensembles. Similarly, network channels are a scarce resource requiring controls on how they may be used.

An economic model can be used to protect against resource consumptive behavior of agents in a multi-agent system [3]. Note that control in agent systems is not based solely on programming structures, as agents may create or invoke other autonomous agents. Such autonomy makes it important to devise explicit mechanisms for controlling the extent to which an expanding group of agents, working on a single task, can utilize a system's resources. In an open distributed system, the problem is compounded by the ability of agents to exist in a resource space not entirely dedicated to their computations alone. We need mechanisms to support bounding the resource utilization of individual agents, or ensembles of agents working together, according to the terms under which they are allowed access to those resources.

Consider a variation of the distributed knowledge acquisition application described earlier, where we want to control the degree of resource consumption in pursuit of the goal. Typical messages to agents will contain values representing resource allocations for servicing them. Every agent has a resource consumption strategy that tells it what portion of the available resources may be allocated to which sub-task. Each agent develops its semantic network only so long as it has sufficient resource allocation, and stops when just enough resources remain for transmitting results back to the client.

To implement an economic model, we will use the notion of a universal currency. Specifically, resource allocation will be measured in a common currency called GCU (for *global currency unit*). Every computational activity must be al-

located GCU's which may be used in completing the task. Each agent is alloted some subsistence GCU's at the time of its creation by its creator, and because activity in message-based systems is triggered by a message, GCU's must also be allocated at the time of sending a message. The GCU's so transferred are deducted from the accounts of the creator or the sender, respectively.

The notion of computational resources must be broad enough to include all entities in the system whose use by one agent can affect the performance of rest of the system. These may include both physical resources (e.g., processors, memory, etc.) and logical resources (e.g., threads). The analog of renting seems to apply more naturally here than that of purchasing.

In addition to the resources consumed while progressing towards accomplishing their goals, individual agents may sometimes be waiting for information from elsewhere, or for reasons of coordination. Such waiting consumes memory resources which must be accounted for. At the same time, an agent should not have to pay if the idle wait is increased by the host's own scheduling choices. Thus, it is important to represent resources both in terms of individual agents as well as in terms of the larger application they are serving at a particular hosting node. Only the delays caused by co-agents in an application should be charged.

Similarly, it is also important to distinguish between economic boundaries in an open distributed system and the physical boundaries between computational nodes. Although resources such as network bandwidth usage depend on physical boundaries, costs of other resources would more logically vary as one crosses economic boundaries.

An agent interested in migrating to a particular host must negotiate a contract with the host ahead of the actual migration. Independent of the actual negotiation protocol, the purpose is to agree on a function that would determine costs of resources that will be made available, possibly dependent on the state of the host. Once the contract has been agreed, an agent may arrive at the node with a certain number of GCU's. The GCU's held by an agent may be spent for purchasing computational resources from the host as it computes, according to the negotiated contract.

The contract between an agent and its prospective host may also decide the granularity at which it would be charged for its activity, which would in turn determine how fine-grained the monitoring would need to be. For example, the host may offer free memory usage with a more expensive CPU cost, removing the need for monitoring memory usage. At the same time such decisions may have important implications. For a node offering free memory usage, an agent may arrive at the node, spawn a large number of child agents who just sit there occupying space. But at the same time, as explained earlier, charging for the time for which memory is in use is non-trivial. An agent may not be charged for staying on a node longer because of delays caused by the node's scheduler. We want to charge the agent only if there is no message in the system for it, for the time that its co-agents are executing. One way in which this would be

possible would be if the host's scheduler would schedule an application scheduler for each application, rather than scheduling individual agents directly. In this way, the rent for the memory being used can be charged only for the time for which the application is scheduled. Not charging for memory may hence be a cost-effective compromise in some cases to avoid excessive overhead involved in monitoring memory usage.

Application specific schedulers enhanced with a scheme to dynamically assign priorities to agents may also be useful in bounding the autonomy of agents, allowing them to cooperate by resource sharing. An agent may fine-tune priorities of its child agents or peer agents may themselves choose to lower their own priorities to allow others, working on more time-sensitive parts of the application, to compute faster.

In addition to the costs of resources provided by hosts, one must also model network bandwidth usage, so that every network communication results in a cost incurred by the initiator, depending on the size of the message and some abstraction of the route to be taken.

2 Agent Ensembles

An individual agents is not much more powerful than a conventional sequential program. However, ensembles of agents exploit parallelism, distribution and mobility to promise orders-of-magnitude greater computational power than conventional programs. But dynamicity and uncertainty in such systems poses a number of problems for the realization of such promise. To allow agent ensembles to operate effectively, we need to provide the ability to organize groups of agents in interesting ways. Specifically, there are three kinds of concerns we have to address. First, the contexts in which the agent ensembles execute and interact need to be dynamically customizable. Second, flexible communication abstractions need to be supported for group level communication. Finally, the interactions of different, potentially overlapping groups of agents, must be mediated to ensure shared protocols. We describe a programming model that provides requisite flexibility.

2.1 Customizing Environments

An agent computing at a node may find the execution environment unsuitable for continued computation, for reasons as varied as a need for large amounts of data available only at a remote site (hence, too expensive to transfer) or real-time guarantees required for executing a specific piece of code. Although such reasons for migration may depend on an agent's functional behavior, it is desirable to separate environmental concerns from the agent's application code. To achieve this separation, we introduce the notion of a *facilitator*.

Every agent's definition includes a dynamic declarative specification of the requirements attributable to its execution environment. There is a facilitator associated with each agent, which is triggered when the agent modifies its specification. The facilitator first determines whether the new requirements can be satisfied at the local host, by examining the agent's status, and possibly renogotiating its host contract. If the host cannot satisfy the agent's new requirements, the facilitator begins negotiations with managers of other nodes (e.g., facilitator for Agent3 in Figure 4). If there a more suitable node is available, the facilitator migrates the agent there.

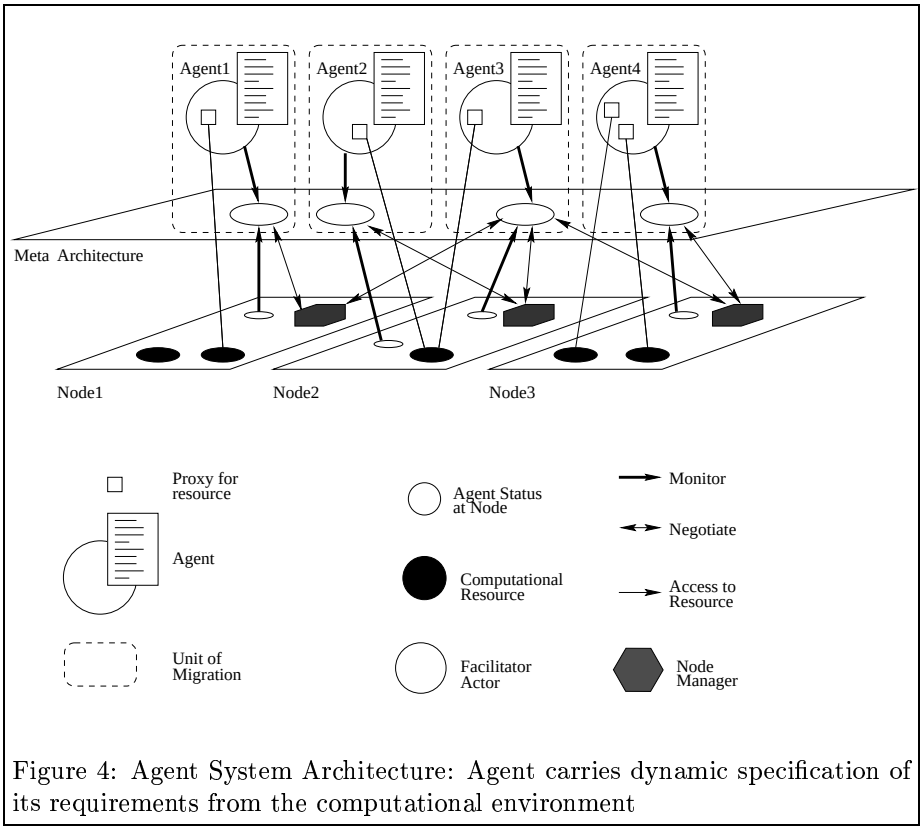

Figure 4: Agent System Architecture: Agent carries dynamic specification of its requirements from the computational environment

Environmental requirements specification of an agent comprises specification of required data, processor time, memory, disk space, and network bandwidth, availability of these resources, and their costs. An agent may also require specialized services from a host: it may need to be mediated, contained, or scheduled to meet requirements such as security, real-time, or Quality of Service (QoS).

The model we use for separating environmental concerns for the application code of agents is *reflection*. Reflection allows an application to monitor

the execution of the underlying system and to modify it dynamically. Specifically, facilitators are part of a meta-architecture enabling agents to continuously interaction with their environment.

Agent migration in this model is transparent to the migrating agent itself. The facilitator is responsible for completing the transfer, and setting up the agent's environment at the new host. This setup may entail obtaining proxy names for the resources needed by the agent, and customizing the environment for requirements such as security, real-time etc. Figure 4 gives a representation of this architecture.

Reflection also enables customizing middleware of the agent's new host. In general, models of reflection enable interaction of higher level requirements, such as real-time constraints, and lower level information about the execution environment, such as load distribution over a group of processors, or available network bandwidth.

Because the Actor model allows the state of the computation to be modeled directly, the computation environment called the *meta-level architecture* can be represented at an appropriate level of abstraction using the same base language [10].

In Rosette [9], a commercially developed object-oriented implementation of an Actor architecture, the architecture has an *interface layer* and a *system environment*. The interface layer provides *mechanisms* for monitoring and control of applications, where the system environment contains actor communities which implement resource management *policies*, providing monitoring, debugging, resource management, system simulation, and compilation/transformation facilities. To support reflection of the interface layer, Rosette uses three classes of resource actors to abstractly implement an actor: *container*, *processor*, and *mailbox*. Containers model the storage local to actors, in a way similar to frames in knowledge-based systems.

2.2 Pattern-Based Communication

We need flexible communication abstractions for programming agent ensembles. An agent may need to communicate with a group of agents or an arbitrary member of a group of agents rather than a particular target agent. In such cases if the sender must name all the potential recipients then a level of abstraction is lost. Moreover, it is often necessary to communicate with agents whose addresses are not previously known. In other words, we need support for a Yellow Pages service to find addresses of agents of a given type. Traders in an object request broker architecture perform a similar function. This necessitates a pattern based naming scheme that identifies agents as being members of groups and allows communication with agents that are not individually known.

The *Actorspace* model allows an abstract pattern-based specification of a group of agents [4]. An actorspace associates an agent with specific attributes. The sender of a message specifies a destination pattern which is matched against

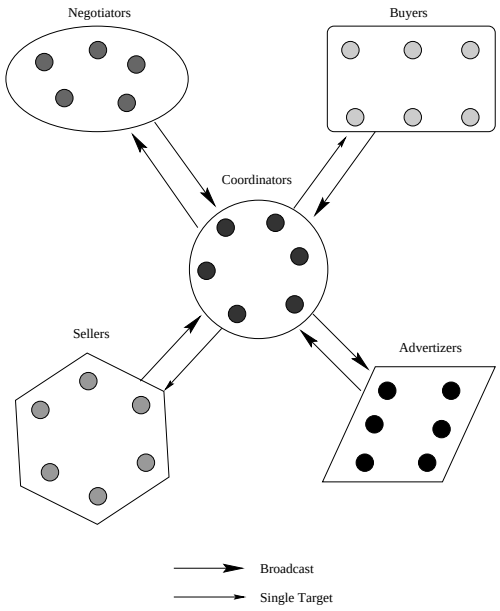

Figure 5: A real estate agency application. There are five actorspaces: coordinators, negotiators, buyers, advertizers and sellers. In all the communications involved the sender need not know the address of the receiver; the sender just specifies the destination pattern which is matched against the attributes of agents in the target actorspace.

the attributes of agents in the actorspace. The sender may send a message to a single arbitrary member of a group, or broadcast it to the entire group. Moreover, the visibility of attributes is dynamic, thus enabling dynamic group membership. Finally, meta-level operations may be associated with an actorspace. For example, an actorspace manager may transparently schedule requests to ensure load balancing. The model may also be seen as providing a distributed version of the *blackboard*[5] system for broadcast communication.

Figure 5 illustrates an example of how actorspaces may be used by a real estate agency. The application consists of five agent groups: coordinators, negotiators, buyers, advertizers and sellers. The membership of each of these groups is dynamic. The coordinator agents make all the business decisions such as which properties to buy or sell, at what price, and when. The negotiator agents constantly look for real estate on sale and report the best deals to the coordinators. The coordinators broadcast their preferences to all the negotiators. These preferences include desired characteristics of the property to be bought, its loca-

tion, the maximum acceptable price, etc. Once a coordinator has decided to buy an estate it must send appropriate instructions to a buyer agent at the location. For this, the coordinator may set the destination pattern of the message to the appropriate location. The buyers that have registered under this pattern, i.e., the ones at the location specified, would then receive the message. A coordinator may also decide to sell estates, in which case it broadcasts instructions to all advertizer agents. The advertizers visit places to meet potential clients and report the best deals to the coordinator. The coordinator then agrees upon the best option and instructs a single seller to sell the estate to the chosen client. Again, pattern directed communication may be used to contact a seller with desired attributes such as someone closest to the client's location. Note that in none of the interactions described above does the sender of a message need to know the actual address of the intended receiver.

2.3 Interaction Policies

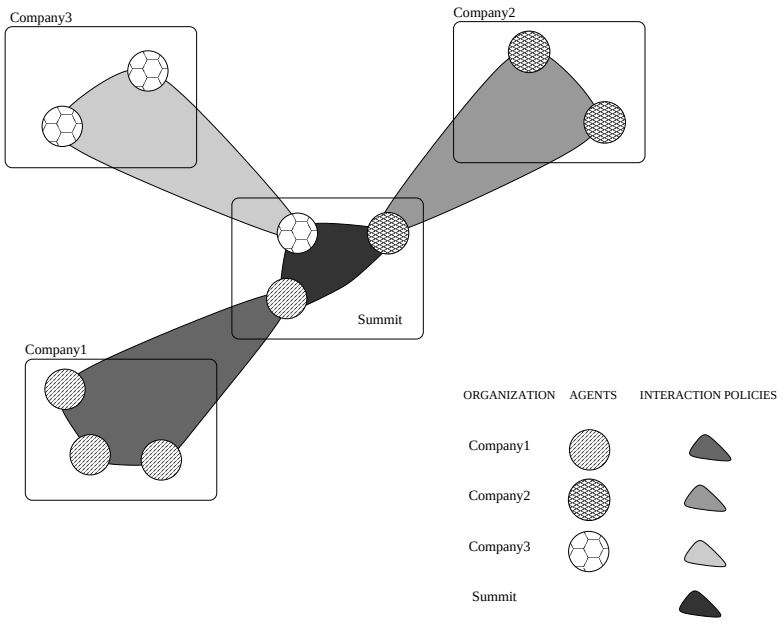

Figure 6: A conference for diplomatic negotiations. The agents at the conference agree upon an interaction policy to communicate with one another. Each agent at the conference in turn uses a specific interaction policy to communicate with the agents back at its company.

An agent application consists of groups of agents carrying out specific tasks and interacting with one another through a set of interaction policies. Different interaction policies may be appropriate in different groups of agents and they may also change with time. Apart from providing communication abstractions, interaction policies may also be used for various other purposes such as fault-tolerance and coordination.

The implementation of interaction policies can be quite involved: it involves exchanging a number of messages between participating agents. Current techniques for developing agent applications require developers to implement interaction policies and individual agent behaviors together, significantly complicating code. The lack of modularity in this approach makes it difficult to reason about code and limits reusability and portability. Moreover, the resulting code is brittle: modifying an interaction policy to satisfy changing requirements requires modifying the code of each relevant component and then reasoning about the entire system, essentially from scratch. Dynamic changes in interaction policies further complicates the situation.

In contrast, meta-level specification of interaction policies through linguistic constructs called *protocols* [7, 8], enables separation of interaction code from agent functionality, and allows dynamic changes in the interaction policies [7]. A protocol is an abstract specification of an interaction policy which can be instantiated on different groups of agents. When instantiated, a protocol governs the interaction between a group of agents by imposing a role on each of the group members. A role, among other things, customizes the behavior of the underlying mail system, and implements one end of an interaction policy. An important advantage is that protocols can be instantiated dynamically and reconfigured by invoking operations defined on them. Moreover, protocols can be composed. Thus, an agent can be in different roles with respect to different protocol instances. The protocol abstraction is realized by a reflective approach described in section 2.1.

Figure 6 illustrates a situation where several groups of agents have different interaction policies imposed on them. Three companies send their representative agents to a summit for diplomatic negotiations. Since the companies do not necessarily trust each other, the summit is held at a trusted brokerage. At the summit the representatives agree upon a common interaction policy. Each representative may in turn use a different interaction policy to communicate with the agents back at its company. For example, each representative may use an encyprition protocol known only to agents of its own company. Note that each agent at the summit has two protocol instances imposed on it and hence is simultaneously in two different roles. Moreover the protocol instances shown are imposed on the agents only for the duration of the summit.

SIDEBAR: Java-Based Agent Systems

There are various implementation issues relevant to agent systems and cur-

rent systems take different approaches in addressing them. When allowing agents to migrate from one node to another, the class definition in the agent's context must somehow be made accessible. Odyssey [3] assumes that the nodes share a common file system and hence there is no need to transfer class files, whereas Aglets [2] and Concordia [1] *serialize* an agent's class definitions along with its state when shipping it to a different node. The mechanics of migration also vary. An aglet may be dispatched or ask to be dispatched to any node running an Aglet context, allowing it to resume its execution at the remote host. Odyssey and Concordia use dynamic itineraries for specifying migrations. Itineraries are lists of host/method pairs which determine the sequence in which an agent must travel from one host to another and the methods to be executed at each host. Itinerary based migration cannot occur in the middle of a method's execution.

Concordia and Odyssey encapsulate a single thread inside every agent, making them closer to actors. Aglets allow multiple threads to execute on a single agent. Communication is only by synchronous message passing in Odyssey, but both asynchronous and synchronous messages are supported in Concordia and Aglets. Aglets preserve the relative order of messages transmitted between any two agents.

None of these systems directly addresses controlling resource consumption of agents arriving at a node. Concordia controls resource access based on *security clearance* of foreign agents, which being a dynamic property based on the user of the agent, can be used for controlling resource usage. In general, agent systems address resource control as a question of whether or not to allow access to certain resources; not to what extent.

In all the above, scheduling is left upto the Java scheduler, which being not standardized, is often not fair, potentially starving some agents.

References

[1] Mitsubishi Electric ITA. Concordia: An infrastructure for collaborating mobile agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Mobile Agents. Proceedings of the First International Workshop, MA '97*, volume 1219 of *Springer Lecture Notes in Computer Science*, Berlin, Germany, April 1997. Springer Verlag.

[2] IBM Tokyo Research Lab. Aglets: Mobile Java Agents. ⟨*http://www.ibm.com.jp/trl/projects/aglets*⟩.

[3] General Magic. Mobile agent white paper. ⟨*http://www.genmagic.com/agents*⟩.

3 Implementing Agent Systems

The authors are developing a prototype based on a Java implementation of Actors called ActorFoundry. The current version of ActorFoundry provides support for basic actor primitives, migration, and a meta-architecture for customizing communication. The primary focus of this effort is to study ways in which agent ensembles behave and mechanisms by which their behaviors can be controlled using the notion of an economy.

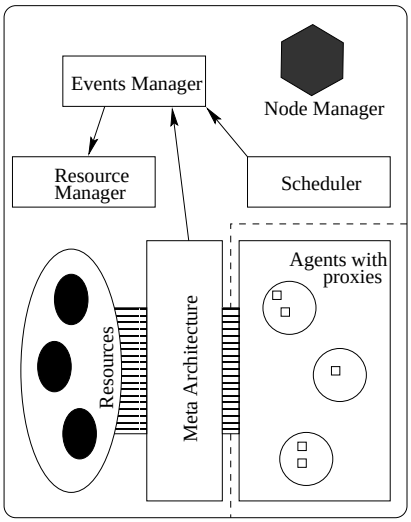

Figure 7: Node architecture

Every node has a node manager that serves as the contact agent for negotiating terms under which an agent may arrive and compute at the node. A contract determines the rate at which an agent would be charged for various resources, possibly depending on the state of the node, and is represented by a list of such functions. Note that there will be separate functions for time/space of usage of a resource, and for the guaranteed needed for the degree of access to the resource.

Nodes support the computation and communication needs of agents through a meta-architecture. Every agent admitted to the system by the node manager is provided names of proxies for the resources it may access; the proxies are customised for each agent according to the negotiated contract. For example, the agent may have specific encryption requirements for communicating with remote agents. This requirement may be satisfied by including an encryption module in the communication proxy. The proxies also address security concerns

of the hosting node.

We use an event based model of communication at the node level to monitor resource usage of agents being hosted by the node. The node scheduler schedules agents for computation according to their contracts, and throws events representing resource consumption by the agents, along with the meta-architecture. The resource manager module subscribes to the events it is interested in, and uses them to update the financial status of agents. The resource manager also marks agents without any GCU's remaining in their accounts as garbage.

Agents are identified by their globally unique names, which remain unchanged as agents move from node to node. When an agent migrates from a node, that node's name table is modified to represent the fact. Communication is both by agent names as well as by advertised behavior patterns. Using a publication/subscription mechanism, agents may advertise the services they are willing to offer, and agents looking for such services would be notified when they become available.

SIDEBAR: Related Work

There are two aspects to programming multi-agent systems – the mechanisms defining an individual agent's behavior (its computational engine), and mechanisms to support coordination between agents. Computational engines of individual autonomous agents in DAI have traditionally piggybacked on advances in conventional AI. In addition, DAI research has addressed issues related to communication and coordination among agents. At the linguistic and system level, a focus of the DAI research has been to provide the abstractions and tools necessary to develop agents. We will call a system providing such linguistic and system level support an agent architecture.

One of the earliest testbeds for building agent architectures was provided by the MACE system [1], which executed in a distributed memory multiprocessing environment. Based on the experience of this research, Les Gasser [2] outlined the avenues of cooperation between the areas of DAI and concurrent programming, and how the two fields can be brought closer to each other. The current proposal draws part of its inspiration from the insights obtained by that research. More recently, an actor-based DAI system called InfoSleuth [10] has been developed at MCC.

The term Agent Oriented Programming has been coined by Shoham [7] to refer to a specialization of Object Oriented Programming (as in actor programming), where the state of an actor (now called an agent) contains beliefs, capabilities, choices and similar *mental* notions, and the computation consists of agents' social interactions with each other, such as informing, offering, accepting, rejecting, competing, assisting, and so on.

A multi-level architecture for Multi-Agent Systems is described by Werner [8]

where a meta-architecture is defined to formalize users', programmers' or designers' interactions with an open system. Michael Kolb's CooL (Cooperation Language) [5] provides a higher level of abstraction with respect to agent design than the actor paradigm, but it gives a knowledge and execution perspective on agents rather than employing mental states. It is possible to give a high level specification of cooperation by negotiating a cooperation object (e.g. goal, plan, schedule) or by synchronizing mutual execution of a plan.

Another context in which the term agent has recently been used is the world wide web (WWW), and there has been an explosion of interest in building agents, in this community too. The use of the term *agent* in DAI and in WWW has different but related meanings. In both contexts, agents are mobile, persistent pieces of code that execute autonomously. In DAI systems, agents may be more complex pieces of code exhibiting intelligence, either individually or collectively; while in the context of the WWW, this is not necessary.

Although Java does provide support for concurrent programming, it is not based on any formal model of concurrency. It allows multiple threads to run concurrently, but unlike actors, Java objects and threads are separate entities, and its passive object model fails to abstract over units of concurrency. The `synchronize` primitive provided for enabling safe usage of concurrent threads is a very low-level facility and its overuse by paranoid programmers often results in deadlocks. This separation of object and thread also creates a problem for migration. By providing Actor primitives in the form of a library, the Actor Foundry [6] developed at OSL attempts to put a discipline for system development in Java.

The Mobile Agent Facility Specification by the Object Management Group [4] makes a case for standardizing areas of mobile agent technology to promote interoperability. These include agent management, transfer, naming (agent as well as agent system), agent system types and location syntax.

Telescript [9] addresses using a public network as a platform on which third-party developers can build their applications. This platform is based on a *remote programming* paradigm that uses Mobile Agents (MA) that can migrate from a client to a remote server and execute remotely on behalf of the client.

Cybenko's group at Dartmouth [3] addresses the issues in implementing mobile agents in an environment consisting of computers, which are often disconnected from the network. Cybenko's mobile agent system, AgentTcl reduces migration to a single instruction, provides transparent communication among agents (hiding all transmission details), and provides a simple scripting language as the main agent communication language while allowing straightforward addition of new languages and transport mechanisms.

References

[1] L. Gasser, C. Braganza, and N. Herman. Mace: A flexible testbed for distributed ai research. In M. N. Huhns, editor, *Distributed Artificial Intelligence*,

pages 119–152. Pitman - Morgan Kaufmann, 1987.
[2] Les Gasser and Jean-Pierre Briot. Object-based concurrent programming and distributed artificial intelligence. In Nicholas M. Avouris and Les Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*, pages 81–107. Kluwer Academic, 1992.
[3] Robert S. Gray. A flexible and secure mobile-agent system. In Mark Diekhans and Mark Roseman, editors, *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL '96)*, Monterey, California, July 1996.
[4] Crystaliz Inc., General Magic Inc., GMD FOKUS, and IBM Corporation. Mobile Agent Facility Specification. Technical report, Object Management Group, June 1997.
[5] Michael Kolb. A cooperation language. In *Proceedings: First International Conference on Multi-Agent Systems*, pages 233–238, San Francisco, CA, June 1995. AAAI, AAAI Press, MIT Press.
[6] Open Systems Laboratory. The actor foundry: A java-based actor programming environment. *Available for download at ⟨http://www-osl.cs.uiuc.edu/foundry.html⟩.*
[7] Yoav Shoham. An overview of agent-oriented programming. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 271–290. MIT Press, 1997.
[8] Eric Werner. The design of multi-agent systems. In Eric Werner and Yves Demazeau, editors, *Decentralized A.I. 3. Proceedings of the Third European Workshop on Modelling Autonomous Agents in a Multi- Agent World, Kaiserslautern, Germany*, pages 3–28. North-Holland, August 1992.
[9] James E. White. Mobile agents. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 437–472. MIT Press, 1997.
[10] D. Woelk, M. Huhns, and C. Tomlinson. InfoSleuth agents: The next generation of active objects. *Object Magazine*, July/August 1995.

4 Conclusions

The development of programming language constructs to allow high-level description of behavior for scalable agent ensembles must await a better understanding of what we need to represent. What is now better understood is how to separate the description of agents functional actions from that of other aspects such as naming, scheduling, and synchronization. These modularity and abstraction mechanisms that have been developed in concurrent programming in general go a long way towards providing the basis for designing and experimenting with powerful agent systems.

References

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

[2] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3–14, May 1993.

[3] Gul A. Agha and Nadeem Jamali. Concurrent Programming for Distributed Artificial Intelligence. In Gerhard Weiss, editor, *Multiagent Systems: A Modern Approach to DAI*, chapter 12. MIT Press, 1998. To appear.

[4] C. J. Callsen and G. A. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.

[5] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and R. D. Reddy. The Hearsay-II speech understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys*, 12(2), 1980.

[6] Munindar P. Singh. *Multiagent Systems.* Number 799 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1994.

[7] D. Sturman and G Agha. A Protocol Description Language for Customizing Failure Semantics. In *The 13th Symposium on Reliable Distributed Systems, Dana Point, California.* IEEE, October 1994.

[8] Daniel C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing.* PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[9] C. Tomlinson, W. Kim, M. Schevel, V. Singh, B. Will, and G. Agha. Rosette: An Object Oriented Concurrent System Architecture. *Sigplan Notices*, 24(4):91–93, 1989.

[10] N. Venkatasubramanian and C. L. Talcott. Reasoning about Meta Level Activities in Open Distributed Systems. In *Principles of Distributed Computing*, 1995.