Appeared in *Multiagent Systems: A Modern Approach to DAI*, Ed. Gerhard Weiss, Chapter 12, Pages 505-534, MIT Press, 1999.

12 Concurrent Programming for Distributed Artificial Intelligence

Gul A. Agha and Nadeem Jamali

12.1 Introduction

The increasing performance and decreasing cost of processors and computer networks has continued to fuel an explosion of interest in solving larger problems using concurrent computing. In particular, agent-based programming has emerged as a promising paradigm which may help realize Artificial Intelligence through distributed problem solving. Agents are persistent and goal directed entities that may move between hosts in response to changes in requirements such as security and efficiency, and that would normally be limited in the computational resources they may employ in pursuing their goals. Such resources include processor time, memory, and network bandwidth.

A key challenge in concurrent computing is the difficulty of programming parallel and distributed architectures. Many models of concurrency are rather low-level. For example, shared variable models often violate data encapsulation, an essential feature for modular software development. A promising approach to address this difficulty is the use of concurrent objects in a reflective architecture. In particular, *actors* provide a formal model for building and representing the behavior of concurrent objects and thus serve as a foundation for concurrent object-oriented programming.

The definition of actors corresponds to that of agents given in Chapter 1. Actors are autonomous, interacting computing elements, which encapsulate a behavior (data and procedure) as well as a process. Different actors carry out their actions asynchronously and communicate with each other by sending messages. The basic mechanism for communication is also asynchronous and buffered; however, other forms of message passing can be defined in the context of the model. Finally, actors may be dynamically created and reconfigured, which provides considerable flexibility in organizing concurrent activity.

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

There are two advantages to using actors for building multi-agent systems. First, actors provide a logically distributed programming model which allows systems to be decomposed into autonomous, interacting components without the need for managing the concurrency explicitly. Second, by using actor implementations on parallel and distributed architectures, performance gains will allow larger problems to be solved.

In this chapter, we discuss a powerful concurrent programming paradigm for DAI; the paradigm is based on abstractions built using extensions of the basic Actor model.

12.2 Defining Multi-Agent Systems

Defining agents has been an elusive problem. A common type of agent is the various personal assistants that have recently become commercially available; such agents 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. Often these agents have the decision making authority to make binding contracts on behalf of a user, such as by purchasing something using a credit card number. Another type of agent uses a variety of filtering mechanisms to make the huge amount of information available over (say) the Internet more manageable for human consumption. All these can be seen as examples of personal agents that act for or on behalf of a user.

A study of personal agents is limited in a fundamental way. Because there is a 1-to-1 correspondence between *interests* and agents, each agent competes or cooperates with others on the basis of its own interest. Although some notion of a "cooperation instinct" can be coded into the interests of agents, it may come at the cost of reduced code re-usability.

A common limitation comes from either not addressing the issue of *mobility*, or not doing so in the context of an open system. In an open system, mobile agents would be able to migrate from one node to another looking for desired computation environments at affordable costs, and to spawn child agents to pursue subtasks. There is no interesting model available to help control the *resources* that such mobile agents serving some particular interest could use. Even in the case of a single node, there is no way of preventing agents pursuing a particular task from monopolizing the entire system's resources.

Let us consider the example of a system of mobile agents spread over a large network, related to the construction industry. There will be agents for clients looking for contractors, agents for contractors looking for potential clients, and agents for smaller sub-contractors at different levels. Each agent shops around and tries to negotiate the best deal for its own interest. But, unless controlled, any number of overly aggressive (say) contractor agents could spawn hundreds of child agents looking for potential clients in parallel, potentially bringing the entire system down. Worse, even well-meaning agents do not have the means to decide what is a reasonable use of the available resources.

Similarly, there is a possibility of multiple child agents working for the same agent (i.e. serving the same interest) to take competing postures. Even if means are provided for some sort of coordination to emerge at a higher level, such agents may still be competing for computational resources at the scheduler level.

These reasons make it important to study ways of controlling ensembles of agents. 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. In the following sections, we will develop a model for studying systems of such agents, that addresses these issues.

We represent agents as actors; specifically, we extend the actor model to explicitly model the location of agents on location on particular *hosts* and the fact that agents have *bounded computational resources*. Hosts are actors that manage physical and logical resources and offer them to agents interested in paying for them. A *uniform currency* is used to pay for the cost of these resources. The behavior of an actor may be interpreted in a suitable framework for agents, e.g., the belief, desire, intent model [28]. In any event, 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. These computational engines include a resource utilization strategy. Of course, all these aspects of an agent may evolve dynamically.

Although the description of goals and procedures falls largely in the domain of conventional AI, explicit resource modeling is a need specific to multi-agent systems. Control is not based solely on programming structures, as agents may create or invoke other autonomous agents. The resource consumption model provides the basis for an economic model that is needed to provide mechanisms to bound use of computational and network resources.

An agent which has a model of its own behavior and that of the environments in which it may be executed, may improve its resource consumption by using mobility. Moreover, because an agent may execute in new contexts which do not satisfy its requirements, the agent may need to systematically customize behavior of the underlying execution environment. Such agent requirements include security, rendering software, device drivers, etc.

A model of *computational reflection* [22] provides a formal basis for an agent to have a representation of its own behavior. In general, reflection models enable interaction of higher level operations, such as real-time constraint enforcement, and lower level information about the execution environment, such as load distribution over a group of processors, available network bandwidth, etc. Specifically, reflection allows an agent to have a continuous interaction with its environment in order to determine available resources and relate such resources to the agents' own state; thus the use of reflection can support evolving resource utilization strategies.

12.3 Actors

Actors are self-contained, interactive, autonomous components of a computing system that communicate by asynchronous message passing [1, 5]. The basic actor primitives are:

- `send`(a, v) creates a new message:
  - with receiver a, and
  - contents v
- `newactor`(e) creates a new actor:
  - which is evaluating the expression e, and
  - returns its address
- `ready`(b) captures local state change:
  - alters the behavior of the actor executing the `ready` expression to b
  - frees that actor to accept another message.

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

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

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

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

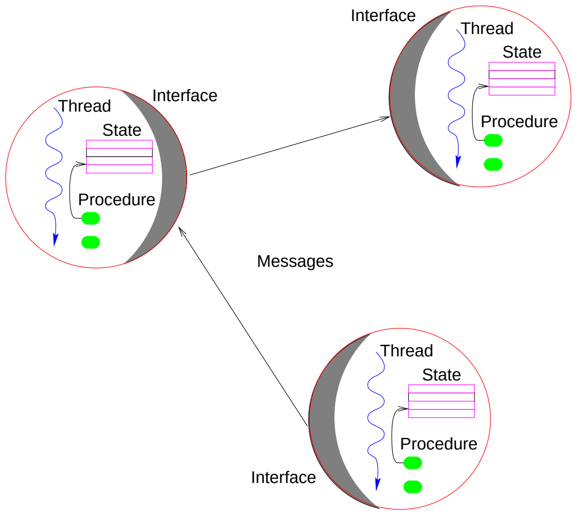

Figure 12.1 Actors encapsulate a thread and state. The interface is comprised of public methods which operate on the state.

Example 12.1 Filtered Search

Consider the problem of a parallel multi-ary tree search, where we want to use a function `filter` to determine what subset of a set of results obtained is useful, before sending them on to the client. There are two different behaviors being defined. FILTERSEARCH has a single method (hence, not named) that takes two parameters, the identity of the customer `cust` and the tree to be searched `tree`. Assume that the number of subtrees and a list of the subtrees can be obtained by using functions `num-children` and `children` respectively; `content` returns the content of the root. After checking for the base case, the behavior FILTERSEARCH creates a *join continuation* actor `jc` with behavior COLLECT with its client's identity `cust` and the number of subtrees `num-children tree` as acquaintances. Next, the actor creates a new actor with its own behavior for each of the tree's subtrees, and sends each new actor the identity of the join continuation actor as its client, and one of the subtrees to search. Once this is done, it gets ready to service another request.

```
(defActor FILTERSEARCH ()
  (let ((filter (lambda (list)
                  ... )))
    (method (cust tree)
      (if (= (num-children tree) 0)
          (send cust (content tree))
          (let ((jc (newActor COLLECT
                        (cust (num-children tree)
                    (list (content tree)) filter))))
          (map (lambda (x)
                   (let ((f (newActor FILTERSEARCH ())))
                        (send f jc x)))
               (children tree))
          (ready FILTERSEARCH ())))))
```

An actor with behavior COLLECT is created with acquaintances `cust`, `n`, and `results` to represent the customer, the number of values to expect, and the list of results collected so far, respectively. After receipt of each new result, the actor gets ready to receive more results with the same behavior acquaintances modified to represent state change. When all results have been received, it uses the function `filter` to eliminate unwanted results, and sends the remaining to its client. Finally, the actor changes into a SINK which ignores all messages.

```
(defActor COLLECT (cust n results filter)
  (method (res)
    (cond ((> n 1)
            (ready COLLECT (cust (- n 1) (append res results))))
          ((= n 1)
            (send cust (filter (append res results)))
            (ready SINK ())))))
```

Here is how a typical FILTERSEARCH actor would be created and invoked.

```
(let ((FS (newActor FILTERSEARCH ())))
     (send FS self tree))
```
□

12.3.1 Semantics of Actors

It is possible to extend any sequential language with the actor constructs described above. For example, the call-by-value λ-calculus is extended in [4].

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

Definition (Actor Configurations): An *actor configuration* with actor map, α, multi-set of messages, μ, receptionists, ρ, and external actors, χ, is written

$$\left\langle \alpha \mid \mu \right\rangle_{\chi}^{\rho}$$

where ρ, χ are finite sets of actor addresses, α maps a finite set of addresses to their behaviors, μ is a finite multi-set of (pending) messages. Let $A = \mathrm{Dom}(\alpha)$, i.e., the domain of α, then:

(0) $\rho \subseteq A$ and $A \cap \chi = \emptyset$,
(1) if $a \in A$, then $\mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$, where $\mathrm{FV}(\alpha(a))$ represents the free variables of $\alpha(a)$; and if $\texttt{<}v_0 \Leftarrow v_1\texttt{>}$ is a message with content v_1 to actor address v_0, then $\mathrm{FV}(v_i) \subseteq A \cup \chi$ for $i < 2$.

For an actor with address a, we indicate its state as $\texttt{[}e\texttt{]}_a$, where it is busy executing e; e represents the actor's current (local) processing state.

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

Definition ($\mapsto$):

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \;\Rightarrow\; \left\langle \alpha, \texttt{[}e\texttt{]}_a \;\middle|\; \mu \right\rangle^{\rho}_{\chi} \mapsto \left\langle \alpha, \texttt{[}e'\texttt{]}_a \;\middle|\; \mu \right\rangle^{\rho}_{\chi}$$

$$\left\langle \alpha, \texttt{[}R[\![\texttt{newactor}(e)]\!]\texttt{]}_a \;\middle|\; \mu \right\rangle^{\rho}_{\chi} \mapsto \left\langle \alpha, \texttt{[}R[\![a']\!]\texttt{]}_a, \texttt{[}e\texttt{]}_{a'} \;\middle|\; \mu \right\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

$$\left\langle \alpha, \texttt{[}R[\![\texttt{ready}(v)]\!]\texttt{]}_a \;\middle|\; \mu, \texttt{<}a \Leftarrow v\texttt{>} \right\rangle^{\rho}_{\chi} \mapsto \left\langle \alpha, \texttt{[app}(v, v)\texttt{]}_a \;\middle|\; \mu \right\rangle^{\rho}_{\chi}$$

$$\left\langle \alpha, \texttt{[}R[\![\texttt{send}(v_0, v_1)]\!]\texttt{]}_a \;\middle|\; \mu \right\rangle^{\rho}_{\chi} \mapsto \left\langle \alpha, \texttt{[}R[\![\texttt{nil}]\!]\texttt{]}_a \;\middle|\; \mu, \texttt{<}v_0 \Leftarrow v_1\texttt{>} \right\rangle^{\rho}_{\chi}$$

$$\left\langle \alpha \;\middle|\; \mu, m \right\rangle^{\rho}_{\chi} \mapsto \left\langle \alpha \;\middle|\; \mu \right\rangle^{\rho'}_{\chi}$$
$$\text{if } m = \texttt{<}a \Leftarrow v\texttt{>},\ a \in \chi, \text{ and } \rho' = \rho \cup (\mathrm{FV}(v) \cap \mathrm{Dom}(\alpha))$$

$$\left\langle \alpha \;\middle|\; \mu \right\rangle^{\rho}_{\chi} \mapsto \left\langle \alpha \;\middle|\; \mu, m \right\rangle^{\rho}_{\chi\cup(\mathrm{FV}(v)-\mathrm{Dom}(\alpha))}$$
$$\text{if } m = \texttt{<}a \Leftarrow v\texttt{>},\ a \in \rho \text{ and } \mathrm{FV}(v) \cap \mathrm{Dom}(\alpha) \subseteq \rho$$

12.3.2 Equivalence of Actor Systems

Based on a slight variant of the transition system described above, a rigorous theory of actor systems is developed in [4]. Specifically, various notions of testing equivalence on actor expressions and configurations are designed and studied. The model provides fairness, namely that any enabled transition eventually fires. Thus fairness implies three things. First, every busy actor eventually makes progress. Second, every actor that is ready to receive a message will eventually receive a message, provided there is a message pending for it. Finally, if an actor does not become "stuck", i.e. is ready infinitely often, it will eventually process every message sent to it. Fairness is an important requirement for reasoning about eventuality properties. It is particularly relevant in supporting modular reasoning: if we compose one configuration with another which has a nonterminating computation, computation in the first configuration may nevertheless proceed as before, for example, if actors in the two configurations do not interact.

The notion of equivalence is defined by adding an observable distinguished *event* to the set of transitions. This technique is a variant of operational equivalence defined in [23]. Two actor expressions may be plugged into a context to see if the event occurs in one or the other case. Two expressions are considered equivalent if they have the same observations over all possible contexts.

The nondeterminism in the arrival order of the messages in an actor computation gives rise to three notions of observation over a computation tree. Notice there are many computational paths in the tree. Now it is possible that the event occurs in every computational path (*must* happen); occurs in some but not all computational paths (*may* happen), or never occurs.

Three distinct well-known equivalence relations may now be defined. In *may* equivalence, *always occurs* is as good as *sometimes occurs* (that is, either is a sufficient condition for proving equivalence); in *must* equivalence *never occurs* is as good as *only sometimes occurs. Convex* equivalence requires the two sets to coincide (the intersection of the two equivalences). An important result is that, in the presence of fairness, the three forms of equivalence collapse to two, namely, *may* and *convex.* Thus, while fairness makes some aspects of reasoning harder – we cannot simply use co-induction in proofs – it simplifies others.

Methods for proving laws of equivalence and proof techniques that simplify reasoning about actor systems have been developed. Finally, the composition of configurations defines an algebra.

Note that the model we have defined thus far does not capture mobility of code. Specifically, λ-abstractions cannot be communicated. Since behaviors are modeled as λ-abstractions, this implies that remote creation and migration cannot be explicitly modeled.

12.3.3 Actors and Concurrent Programming

In addition to the asynchronous message passing paradigm used by the Actor model, other paradigms have also been used for implementing concurrent systems. A detailed treatment of these can be found in [6]. In the *shared variable* paradigm, processes communicate by writing to and reading from memory locations shared by them. Although the apparent simplicity of this paradigm is appealing, it violates principles of abstraction and encapsulation, making it difficult to implement large systems reliably. Among the issues such implementations have to address include support for *mutual exclusion*, the ability to disallow all but one process to access a set of shared variables, and *condition synchronization*, requiring that a piece of code in some process be not executed until some condition is met.

A classic problem in concurrent programming is called the *critical section problem*, in which n processes execute indefinitely long alternating between sections of code that do and those that do not access some shared variables. The part of code that does access these variables is called the *critical section.* The objective is to provide mutual exclusion, while preventing deadlock/livelock or an unnecessary delay, and ensuring that every process attempting to enter its critical section does eventually do so.

A construct that can be used to solve the critical section problem and many other synchronization problems for shared variable systems is *semaphores.* Semaphores provide a disciplined way for supporting condition synchronization by using values of shared counters to control whether a section of code can or cannot be executed. *Conditional critical regions* are an abstraction that groups together shared resources and allows only conditional access to such groups. *Monitors* abstract this further by limiting access to shared variables strictly through use of a fixed set of procedures.

The actor 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. Actors thus provide an abstract level at which to program and reason about agents. Synchronous communication and other more complex communication mechanisms can be built on top of the basic asynchronous communication mechanism [5]. Moreover, as we will see later in this chapter, high-level commit protocols can be used for agent-level synchronization.

12.4 Representing Agents as Actors

In developing multi-agent systems, a key issue to be addressed is *mobility.* Mobility allows an agent to migrate from one node in the distributed system to another, seeking a “better” execution environment. The increased flexibility raises some other important issues.

It may be desirable for an agent to migrate to a different physical location for a variety of reasons. These reasons may include lower cost of execution compared to the current location, or improved quality of service. The need to migrate can also be task specific. For example, if an agent needs to access huge amounts of data at different locations, it may make sense to migrate to those locations in order to exploit better locality.

The above examples essentially assume that mobile agents are clients. On the other hand, it is also possible to have server agents that roam around the network looking for hospitable execution environments attempting to sell their services. This may even take the shape of a partnership whereby server agents are allowed to exist on nodes, and the nodes can advertise the additional services thus made available to attract other clients.

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.

A complementary issue to limiting the resources consumed by an agent is that of supporting an agent or an ensemble of agents in pursuit of their goals. The system must provide means for agents serving the same interest to cooperate, or otherwise not impede each other's progress.

12.4.1 Mobility of Actors

Because Actor semantics is location transparent, systems based on the model (e.g., [18]), do not allow actors to reason about their locations. This limits the use of migration to system level decisions where only system level goals such as load balancing can be considered. To take advantage of agents' ability to autonomously decide whether, when and where they want to migrate, we need to extend the Actor model with notions of location and mobility.

A precursor to true migration is the ability to create an actor at a remote site. The Actor programming language Hal [3] uses annotations to govern actor placement at creation.

Example 12.2 Distributed Filtered Search
Consider a variation of the Filtered Search example we saw earlier, where the tree is distributed over many nodes. New actors for searching the subtrees are created at nodes hosting the roots of the respective subtrees. We assume that the tree is non-empty.

```
(defActor FILTERSEARCH ()
  (let ((filter (lambda (list)
                  ... )))
    (method (cust tree)
      (if (= (num-children tree) 0)
          (send cust (content tree))
          (let ((jc (newActor COLLECT
                      (cust (num-children tree)
                         (list (content tree)) filter))))
                (map
                  (lambda (x)
                    (let ((f (newActor FILTERSEARCH ())
                                         @ (host-of x)))
                       (send f jc x)))
                    (children tree))
                (ready FILTERSEARCH ()))))))
```
□

A similar construct, called `trojan-multisend`, sends new actors to a collection of remote locations, along with the first messages that each will process [3].

True migration must allow an actor to migrate to a different node while it is in the middle of its execution. We will describe a specific way of providing this functionality.

First, we define some important changes in the actor naming scheme that is used, to allow migration to be represented. Because actors can migrate, we need to identify an actor's current location. Specifically, we change the naming scheme for identifying individual actors for the purpose of sending messages: an actor name is now $h.a$, where a is a globally unique identifier for any actor, and h identifies the node at which it currently resides. The important implication is that a name a at any node in the system corresponds to the same actor. Practical implications of the new name representation will be discussed shortly.

The message send that simply resulted in creation of a message from the standard Actor semantics, now creates such a message locally in the host node's queue, necessitating keeping track of which node a message is physically located at, at any time. The transfer of a message from its current location to the target actor's node is handled separately.

Migration can be represented in two ways: the agent language can provide a migration primitive, or it could provide an agent with a way to grab its own state and send it over (inside a message) to a remote node to create a duplicate with that state; the original actor can then become a forwarder. Because a migration primitive introduces greater semantic complexity, we choose to study the latter. A `ccf` primitive can be introduced to grab the local state of an actor by enclosing the actor's reduction context inside a λ-abstraction. Using this primitive, we can represent higher level operations as macros.

Example 12.3 Migration
A construct for migration, called `migrate@h`, may be defined as a macro. Without loss of generality, assume that each host also has a manager actor $h.m$ that acts on behalf of the host and manages the host's resources. The `ccf` primitive is used for grabbing the current continuation of the actor. Unlike Lisp/Scheme, here continuations are local to a single actor; in Scheme, the continuation represents the state of the entire sequential program – typically a much larger object. The function given to `ccf` first sends a `move` request to the remote host's manager `h.m` to create a new actor with the same personal name as the actor requesting migration, using the reduction context enclosed in `y` as its behavior. It then changes the requesting actor's behavior to WAIT-ACK. Assume we have a procedure `getkey` to generate a new key every time it is invoked; `personal-name` returns the name of an actor minus the host's identifier.

```
(let ((k (getkey)))
  (ccf (lambda (y)
         (seq (send h.m move self y k (personal-name self))
              (ready WAIT-ACK (h m k (personal-name self)))))))
```

Assuming that a `move` method is a part of `h.m`'s behavior, it would accept the message, create a new duplicate actor, and return an acknowledgment. The behavior WAIT-ACK waits for an acknowledgment from the remote host manager, containing identities of the host and its manager, and a copy of the key sent with the request.

```
(defActor WAIT-ACK (h m k a)
  (lambda (ret-h ret-m ret-k)
    (if (and (= h ret-h) (= m ret-m) (= k ret-k))
        (ready FORWARDER (h.a))))))
```

To avoid the blocking semantics, the actor may add the method WAIT-ACK to its current behavior rather than replacing with it. In such a case, until the acknowledgment message is received the actor would keep acting as usual. Once the message is received, it would change its behavior into that of a FORWARDER.

Note that because actor names are globally unique, there is no need to transmit the complete name of the new actor. If a particular name is in use at multiple nodes, only one of them corresponds to an actual actor; others have to be forwarders. An important implication of this is that if an actor migrates to a node where the name is already in use, it must be in use as a forwarder which can be safely overwritten by the actual actor. □

Example 12.4 Remote Creation

A construct for remote creation, `remote-actor(e)@h`, may similarly be defined as a macro. As above, assume that `h.m` is the manager actor for the host `h`. Here, the function given to `ccf` sends a `newactor` request to the remote host's manager `h.m` to create a new actor with name `a` and behavior `e`, and changes the requesting actor's behavior to WAIT-ADDR.

```
(let ((k (getkey)))
  (ccf (lambda (y)
         (seq (send h.m newactor self e k)
              (ready WAIT-ADDR (h m y k))))))
```

Behavior WAIT-ADDR waits for the address of the new actor created remotely, and after verifying all the information, it inserts the new address in the reduction context contained in `y`.

```
(defActor WAIT-ADDR (h m y k)
  (lambda (ret-h ret-m ret-k a)
    (if (and (= h ret-h) (= m ret-m) (= k ret-k))
        (ready (y h.a)))))
```

To avoid the blocking semantics in this case, the actor could perform a local `newactor` to create a local actor with a migrate expression preceding rest of the desired behavior. This would be facilitated by the fact that actor names do not change as actors move from node to node. □

Semantics of Mobile Actors

Transitions presented in Section 12.3.1 can now be modified to address support for migration. To identify an actor's current location, a superscript is added to the actor state representation that identifies the host; recall that the subscript identifies the actor itself.

Definition ($\mapsto$):

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \langle \alpha, \texttt{[}e\texttt{]}^h_a \mid \mu \rangle^\rho_\chi \mapsto \langle \alpha, \texttt{[}e'\texttt{]}^h_a \mid \mu \rangle^\rho_\chi$$

We assume that all creation is local and that only messages co-located on the same host as an actor are consumed. Remote messages, actor migration, and remote creation will be dealt with separately.

We keep track of which node a message is physically located at by attaching a superscript to each message, identifying the host of the intended recipient. A separate transition is added to represent the transfer of a message from its current location to the target node.

$$\langle \alpha, \texttt{[}R[\![\texttt{newactor}(e)]\!]\texttt{]}^h_a \mid \mu \rangle^\rho_\chi \mapsto \langle \alpha, \texttt{[}R[\![h.a']\!]\texttt{]}^h_a, \texttt{[}e\texttt{]}^h_{a'} \mid \mu \rangle^\rho_\chi \qquad a' \text{ fresh}$$

$$\langle \alpha, \texttt{[}R[\![\texttt{ready}(e)]\!]\texttt{]}^h_a \mid \mu, \texttt{<}a \Leftarrow v\texttt{>}^h \rangle^\rho_\chi \mapsto \langle \alpha, \texttt{[app}(e, v)\texttt{]}^h_a \mid \mu \rangle^\rho_\chi$$

$$\langle \alpha, \texttt{[}R[\![\texttt{send}(h_2.a_2, v)]\!]\texttt{]}^{h_1}_{a_1} \mid \mu \rangle^\rho_\chi \mapsto \langle \alpha, \texttt{[}R[\![\texttt{nil}]\!]\texttt{]}^{h_1}_{a_1} \mid \mu, \texttt{<}h_2.a_2 \Leftarrow v\texttt{>}^{h_1} \rangle^\rho_\chi$$

$$\langle \alpha \mid \mu, \texttt{<}h_2.a \Leftarrow v\texttt{>}^{h_1} \rangle^\rho_\chi \mapsto \langle \alpha \mid \mu, \texttt{<}h_2.a \Leftarrow v\texttt{>}^{h_2} \rangle^\rho_\chi$$

The two transitions for interaction with actors outside the system remain unchanged, except for the fact that the receptionists ρ and the external actors χ now contain actors as well as host managers.

The following last transition provides access to the local state of an actor, which is needed to support migration. It introduces the primitive `ccf`, which grabs the continuation by putting the reduction context R inside a λ-abstraction, and applying it to the given function v.

$$\langle \alpha, \texttt{[}R[\![\texttt{ccf}(v)]\!]\texttt{]}^h_a \mid \mu \rangle^\rho_\chi \mapsto \langle \alpha, \texttt{[app}(v, \lambda x.R[\![x]\!])\texttt{]}^h_a \mid \mu \rangle^\rho_\chi$$
$$x \notin \mathrm{FV}(R[\![\texttt{nil}]\!])$$

Although these semantics explain the process of migration, note that to establish the need to migrate, an agent must be able to observe its own state. The model of *computational reflection* provides a formal basis for an agent to have a representation of its own behavior. We will discuss reflection in Section 12.5.1.

12.4.2 Resource Model

Resource allocation in multi-agent systems is a problem that raises issues of reciprocity as well as performance and security concerns. Nodes in a multi-agent system over the worldwide 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.

We may use an economic model to protect against resource consumptive behavior of agents in a multi-agent system. Recall 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.

Example 12.5 Bounded Distributed Filtered Search
Consider a variation of the Distributed Filtered Search application described earlier, where we want to control the amount of resources that can be consumed in pursuit of the goal. The typical message send to an actor with behavior FILTERSEARCH will contain a value `res` representing the resources allocated for the task:

```
(let ((FS (newActor FILTERSEARCH ())))
     (send FS self tree res))
```

The system will strip the value `res` from the message, and keep track of the resources remaining at any time. The agent would have read access to the current value of this quantity by asking the system.

The application keeps creating new agents to search subtrees as long as it has resources, and stops when only `delta` remains. We assume that `delta` represents sufficient resources for transmitting results to the client. `part` represents the agent's consumption strategy that tells it what portion of the available resources may be allocated to a sub-task.

Because there isn't a way to know how many messages `jc` should expect at the time of its creation, its initial behavior is set to TELLCOLLECT, which waits for a count of the number of responses to expect. After receiving that message, a TELLCOLLECT actor uses the value in replacing its behavior with COLLECT.

```
(defActor FILTERSEARCH ()
  (let ((filter (lambda (list)
                    ... )))
    (method (cust tree)
      (if (= (num-children tree) 0)
          (send cust (content tree))
          (let ((jc (newActor TELLCOLLECT ()))
                (count 0))
            (map
              (lambda (x)
                     (if (> (my-resources) delta)
                         (let ((f (newActor FILTERSEARCH ())
                                                 @ (host-of x)))
                            (send f jc x (part (my-resources)))
                            (setf count (+ count 1)))))
                   (children tree))
                 (send jc cust count (content tree)
                     (part (my-resource)) filter)
                 (ready FILTERSEARCH ())))))
```

This example does not account for resources needed for agents to survive on a node while they are inactive.

Note that an agent's resource consumption strategy is independent of the system's ability to *pull the plug* when the resources run out. Needless to add, any attempt to send more resources to another agent than it possesses, would be trapped by the run-time system. □

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 would be allocated some GCU's which can be used in completing the task. Because activity in message-based systems is triggered by a message send, these GCU's can be allocated at message send time. But note that what is counted as resources is the physical and logical computational resources needed to service a message. This is not the only use of resources; agents residing at a host waiting for something to happen, for instance, also use resources such as memory. Thus, 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. The amount of time devoted to an agent by the processors, the memories, the disks and the channels, are all resources which need to be paid for. The analog of renting resources seems to apply more accurately 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 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 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.

Semantics of Resource Bounded Agents

In developing the semantics for representing resource allocation, we add a value r to the agent state (we now use the term *agent* instead of actor), to represent the units of universal currency (GCU's) available to the agent. The configuration also includes β to represent the system map, which includes all the *host agents* representing the nodes, and the network connecting these nodes. $[s]_h$ says that the host agent h has state s. We treat the host agents separately because they are not mobile, and because the fact that a host's state may determine the cost of its computational resources, makes it important to keep track of its state changes.

We are also introducing two new functions. T_{st} is a function that takes the current state of a host and the transition being applied, to give the next state. This function is applied to all members of β being effected by a transition. $T_{res(a,h)}$ is a function that represents a contract between an agent a and the node h hosting it, and determines the cost (in GCU's) of performing a transition t when the host is in state s ($\overset{t}{\mapsto}$ will be used to represent transitions, where t is a variable representing the specific transition taking place). Such a contract would be reached at after a process of negotiation between the agent and the host. Note that this function is very general because it allows the cost of the services to vary as the host's state changes.

Definition ($\mapsto$):

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow$$

$$\left\langle \alpha, [e, r]_a^h \mid \beta, [s]_h \mid \mu \right\rangle_\chi^\rho \overset{t}{\mapsto}$$

$$\left\langle \alpha, [e', r - T_{res(a,h)}(t,s)]_a^h \mid \beta, [T_{st}(t,s)]_h \mid \mu \right\rangle_\chi^\rho$$

$$\text{if } r \geq T_{res(a,h)}(t,s)$$

The transitions for `newactor` and `ccf` expressions remain identical to those for mobile actors, except that the actor is charged for the cost of performing the transitions.

Because it is the `send` primitive that initiates a new activity, a certain number of GCU's has to be sent along with the message for pursuing the activity. So, in addition to the cost of the transition, the wealth of the sending actor is also reduced by r'. As the activity is entirely local, only the local host's state changes.

The complementary activity of transferring a message from one node to another represents change in states of both the nodes as well as the state of the network s_n. We make a convention that the cost of this transfer is always incurred by the sender. Because of this, it is important to identify the sender of the message, for which we add a subscript to the messages to represent the sender's identity. The only amount charged for transferring a message is the network cost $T_{net}(h_1, h_2, \mid v \mid, s_n)$ of transferring a message of size $\mid v \mid$. We assume that any cost of handling the message at both ends of the channel is negligible in comparison and can be ignored.

$$\langle \alpha,\, [R[\![\texttt{send}(h_2.a_2, v, r')]\!], r]^{h_1}_{a_1} \mid \beta,\, [s]_{h_1} \mid \mu \rangle^{\rho}_{\chi} \overset{t}{\mapsto}$$
$$\langle \alpha,\, [R[\![\texttt{nil}]\!], r - r' - T_{res(a,h)}(t,s)]^{h_1}_{a_1} \mid \beta,\, [T_{st}(t,s)]_{h_1} \mid \mu, m \rangle^{\rho}_{\chi}$$
$$m = \texttt{<}h_2.a_2 \Leftarrow \texttt{[}v, r'\texttt{]>}^{h_1}_{h_1.a_1}$$

$$\langle \alpha,\, [e, r]^{h_1}_{a_1} \mid \beta,\, [s_1]_{h_1}, [s_2]_{h_2}, [s_n]_{net} \mid \mu, \texttt{<}h_2.a_2 \Leftarrow \texttt{[}v, r'\texttt{]>}^{h_1}_{h_1.a_1} \rangle^{\rho}_{\chi} \overset{t}{\mapsto}$$
$$\langle \alpha,\, [e, r - T_{net}(h_1, h_2, \mid v \mid, s_n)]^{h_1}_{a_1} \mid \beta,\, [T_{st}(t, s_1)]_{h_1}, [T_{st}(t, s_2)]_{h_2},$$
$$[T_{st}(t, s_n)]_{net} \mid \mu, \texttt{<}h_2.a_2 \Leftarrow \texttt{[}v, r'\texttt{]>}^{h_2}_{h_1.a_1} \rangle^{\rho}_{\chi}$$

Receipt of a message simply results in addition of the GCU's sent in the message to the wealth of the receiving agent.

$$\langle \alpha,\, [R[\![\texttt{ready}(e)]\!], r]^{h}_{a} \mid \beta,\, [s]_{h_1} \mid \mu, \texttt{<}h_1.a_1. \Leftarrow \texttt{[}v, r'\texttt{]>}^{h}_{h_2.a_2} \rangle^{\rho}_{\chi} \overset{t}{\mapsto}$$
$$\langle \alpha,\, [\texttt{app}(e, v), r + r' - T_{res(a_1,h_1)}(t,s)]^{h}_{a_1} \mid \beta,\, [T_{st}(t,s)]_{h_1} \mid \mu \rangle^{\rho}_{\chi}$$

Following are the two transitions representing communication with the outside world in the form of transfer of a message to or from the system. Because the cost of such a transfer is to be incurred by the sender, there is no need to represent a cost in the transition when a message is received from outside the system. Transferring a message out of the system does result in a cost that will be incurred by the sending agent. The host state changes occur in the network, the local host h_1, and in the host h_2 of the external actor, but because the external host is itself not included in β, its state change is not represented in the transition.

$$\langle \alpha,\, [e, r]^{h_1}_{a_1} \mid \beta,\, [s]_{h_1}, [s_n]_{net} \mid \mu, \texttt{<}h_2.a_2 \Leftarrow \texttt{[}v, r'\texttt{]>}^{h_1}_{h_1.a_1} \rangle^{\rho}_{\chi} \overset{t}{\mapsto}$$
$$\langle \alpha,\, [e, r - T_{net}(h_1, h_2, v, s_n)]^{h_1}_{a_1} \mid \beta,\, [T_{st}(t,s)]_{h_1}, [T_{st}(t, s_n)]_{net} \mid \mu \rangle^{\rho'}_{\chi}$$
$$\text{if } h_2.a_2 \in \chi, \text{ and } \rho' = \rho \cup (\mathrm{FV}(v) \cap \mathrm{Dom}(\alpha))$$

$$\langle \alpha \mid \beta,\, [s]_{h_1}, [s_n]_{net} \mid \mu \rangle^{\rho}_{\chi} \overset{t}{\mapsto} \langle \alpha \mid \beta,\, [T_{st}(t,s)]_{h_1},$$
$$[T_{st}(t, s_n)]_{net} \mid \mu, \texttt{<}h_1.a_1 \Leftarrow \texttt{[}v, r'\texttt{]>}^{h_1}_{h_2.a_2} \rangle^{\rho}_{\chi \cup (\mathrm{FV}(v) - \mathrm{Dom}(\alpha))}$$
$$\text{if } h_1.a_1 \in \rho \text{ and } \mathrm{FV}(v) \cap \mathrm{Dom}(\alpha) \subseteq \rho$$

Finally we need a transition rule to represent the cost of an inactive agent residing at a host. As explained earlier, this cost is complicated by the fact that we do not want to charge an agent if the wait is caused by factors associated purely with the host itself. Essentially, we want to charge the agent if there is no message in the system for it, for the time that its co-agents are executing. This would make sense 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.

$$\Big\langle \alpha,\, [R[\![\texttt{ready}(e)]\!], r]^{h_1}_{a_1} \;\Big|\; \beta,\, [s]_{h_1} \;\Big|\; \mu \Big\rangle^{\rho}_{\chi} \overset{t}{\mapsto}$$

$$\Big\langle \alpha,\, [R[\![\texttt{ready}(e)]\!], r - \epsilon]^{h_1}_{a_1} \;\Big|\; \beta,\, [T_{st}(t, s)]_{h_1} \;\Big|\; \mu \Big\rangle^{\rho}_{\chi}$$

if $<h.a \Leftarrow [v, r']>^{h_2}_{h_2.a_2} \notin \mu$ for any v, r', a_2 and h_2

and t is a transition in some co-agent of a_1

12.5 Agent Ensembles

Individual agents are not much more powerful than conventional sequential programs. However, by exploiting parallelism, distribution and mobility, ensembles of agents promise orders of magnitude greater computational power than conventional programs. Before the promise can be realized, the dynamicity and uncertainty in such systems poses a number of problems. To allow agent ensembles to operate effectively, we need to provide the ability to organize groups of agents in interesting ways. Specifically, there are two kinds of concerns we have to address. First, the contexts in which they execute and interact need to be dynamically customizable. Second, the interactions of different, potentially overlapping groups of agents, need to be mediated to ensure shared protocols. We describe the programming model that has been developed to provide the requisite flexibility.

12.5.1 Customizing Execution Contexts

An agent traveling from node to node seeking affordable resources may find itself in environments that by default do not meet some of its requirements for execution. For example, an agent may need some helper agents that could be asked to perform specialized tasks, as is the case with a library of plug-ins. In order to ensure the appropriate execution context, the agent could ensure that an acceptable context already exists at the host before migrating there. Alternately, it could customize the context at arrival.

In some cases, the execution of an agent needs to be mediated, contained, scheduled, etc., to meet requirements such as security, real-time, or Quality of Service (QoS). Because the implementation of such requirements is dependent on the physical and logical resources available, the underlying architecture supporting agents must be customizable. It is essential for the ability to customize the execution context that the code for requirements such as QoS be implemented separately from the code for the application functionality. For example, if the agents encoding an

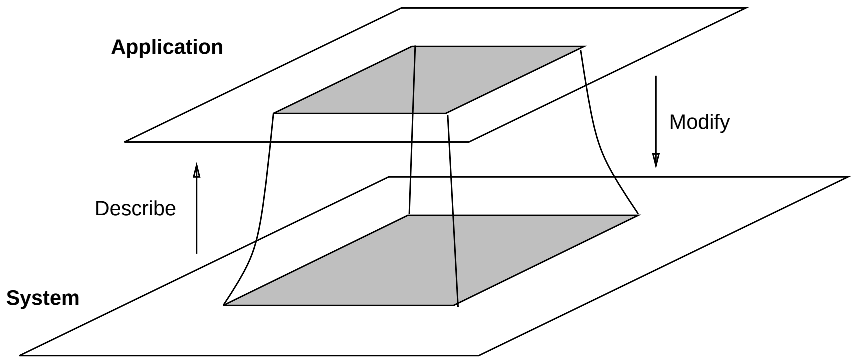

Figure 12.2 Computational Reflection

application are assigning their own priorities and schedules, it is not very feasible to schedule them in order to satisfy real-time requirements.

Customization of the execution context is accomplished using a technique called *reflection* [22]. Reflection allows an application to monitor the execution of the underlying system and to modify it dynamically (Figure 12.2).

Reflection

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

Because the Actor model allows the state of the computation to be modeled directly, the computation environment called the *meta-level architecture* can be represented at an appropriate level of abstraction using the same base language [32]. Specifically, this allows use of reflection enabling an agent to have a continuous interaction with the environment to determine available resources and relate it to its own state to provide evolving resource consumption strategies.

In Rosette [31], 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. Each container is a set of associations (*slots*) of keys with values, which are both other actors. Additions and deletions of slots model allocations and deallocations of storage. Processor actors determine how to determine the method for responding to a message. Mailbox actors buffer incoming messages until they can be processed.

Suppose we want to ensure the availability of some agent where its absence may be catastrophic. We may replicate the service to ensure availability when the original server fails. In the following example adapted from [2], we will see how such a replication service may be provided.

Example 12.6 Replicated Service
We can use meta-actors called *dispatchers* to trap out-going messages, and *mail-queue* meta-actors to trap in-coming messages, for every actor. When a service request arrives for the server, its dispatcher can forward a copy of the request to the alternate servers too. When the servers respond with results, their responses are tagged with an identifier for the request. At the client end, the mail-queue meta-actor can use the tag to discard extra copies of any response. A manager in charge of replicating a service takes the following actions to achieve the state shown in Figure 12.3:

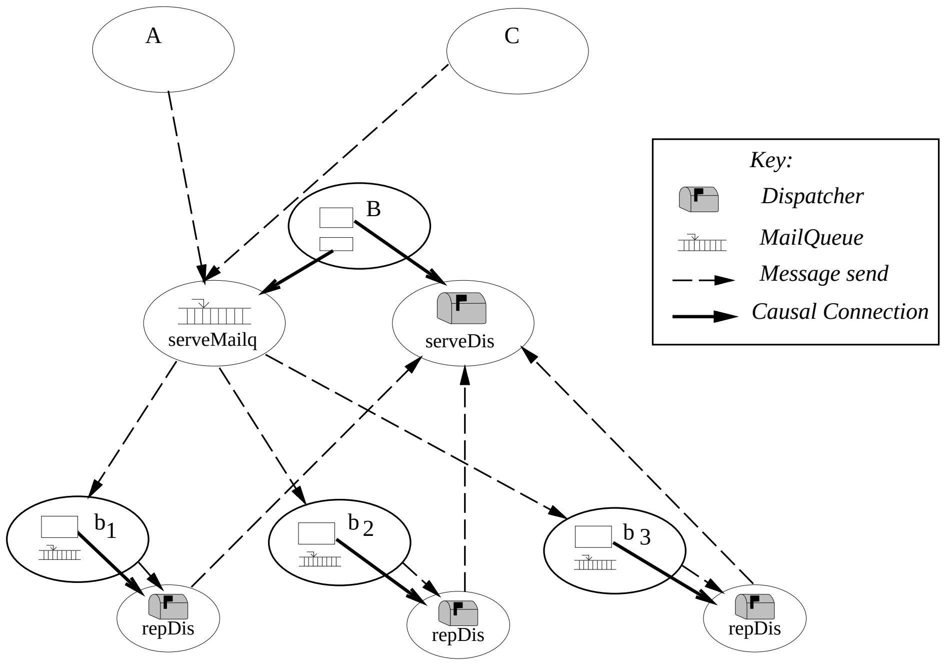

Figure 12.3 When a message is sent by the clients A or C to the replicated service B, the message is received by B's mail queue serveMailq (1). The message is then sent to each of the replicas (2).

1. The specified server is replicated by a manager by creating actors with the same behavior and acquaintance list.
2. A mail queue is installed for the original server to make it act as the *distributor* described above. Messages destined for the original server are broadcast to the replicas. A broadcast using `ssends` (synchronous sends) is done so that all replicas receive messages in the same order and thus solve the same task.
3. The dispatcher of the original server is modified to act as the *collector* described above. The first message out of each set of replica responses is selected to be passed to the destination.
4. The dispatchers of the replicas are changed to forward all messages being sent to the original server's dispatcher. In addition, the messages are tagged so that the original server's dispatcher can eliminate multiple copies of the same message.

The new mail queue for the original server is described using the following behaviors:

```
(defActor SERVEMAILQ (data members)
  (method get (who)
     ... )
  (method put (msg)
     ;; A bcaster actor broadcasts msg to members
     (bsend (newActor bcaster msg) members)))

(defActor BCASTER (msg)
  (method (l)
     (if (not (null? l))
         (ssend (car l) msg)
         (send self (cdr l)))))
```

Note that message order is being preserved in the broadcast. We use `ssend` function to guarantee consistent state at each replica. `bsend` is a remote procedure call (blocking send). Figure 12.3 shows the resulting actions occurring when a message is sent to the replicated service. The original server is actor B. When a message is received by the distributor, `serveMailq` (B's new mail queue), the message is broadcast to the replicas b_1, b_2, b_3. Each of the replicated actors has the same base-level behavior as B. Therefore, upon receipt of the message, each b_i responds in the same way B would have. However, if the replicas respond to the message, the message destinations would be rerouted by the dispatchers $repDis$ to the original server's dispatcher, $serveDis$ (serving as the collector). For each response, $serveDis$ gets three messages, one from each replica. It processes the three messages and sends out a single response to the original destination. Note that the base-level actor B does not receive any messages now since all the incoming messages are redirected to the replicas by its mail queue $serveMailq$ and the outgoing messages are sent by the dispatchers of the replicas directly to its dispatcher $serveDis$. □

12.5.2 Interaction Protocols

Ability of the system to cope with new kinds of failures of a few nodes or parts of the network is essential in a distributed system. A variation of the problem appears when we are dealing with systems where "failure" is the norm, such as distributed systems using wireless communications where the network connectivity is essentially dynamic [15].

When introducing mechanisms for fault-tolerance, it is important to separate the fault-tolerance aspects of the code from the application for reasons of modularity and reusability. In this section we will discuss an abstraction over the primitive Actor model called *interaction policies*. Interaction policies determine what protocols to use in dealing with a failure situation.

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

Notice that the implementation of such protocols can be quite involved: it involves exchanging a number of messages between participating actors. Current techniques for developing distributed software require developers to implement interaction policies and application behavior together, significantly complicating code. The lack of modularity not only makes it hard to reason about code; it limits its 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.

In the first place, in standard programming models, we cannot even express an interaction protocol as a program module; to do so we require the ability to write meta-programs with distributed scope. An interaction protocol imposes a role for each actor, specifically, trapping and tagging incoming and outgoing messages to implement the protocol. Such customization of an individual actor's mail system may be further limited only for the duration of an interaction.

Sturman and Agha have developed a language for describing and implementing interaction policies [29, 30]; using this language, a protocol abstraction may be instantiated by specifying a particular group of actors and other initialization parameters. The runtime system must then support specific forms of reflection, which are sufficiently powerful to enable dynamic modification of the mail system and to store and retrieve actor states, or other parts of the meta-architecture.

Now notice that the semantics of actor systems in the presence of protocols is quite different from the semantics of ordinary (the so-called base-level) actor systems. Our pragmatic experience suggests that reasoning about distributed applications is simplified by our meta-programming system; after all, code size is reduced by at least an order of magnitude, and the application is decomposed into more intuitive units corresponding to the requirements specification. However, the semantics of meta-level operations remains poorly understood. Recent research based on ac-

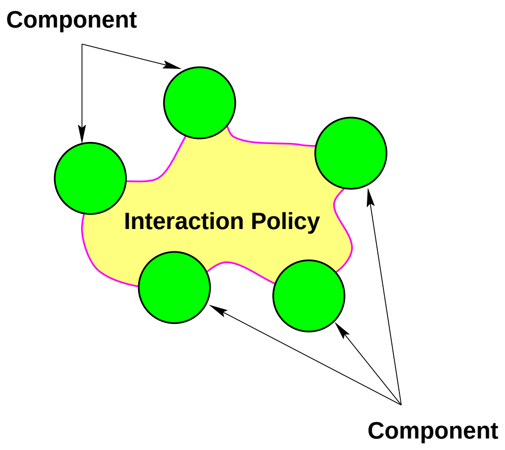

Figure 12.4 A distributed system consists of a set of components carrying out local computations and interacting in accordance with a set of *policies*.

tors has made progress on the problem of reasoning in the presence of meta-actors, specifically, by defining a reasoning system and using it to prove the correctness of a meta-level algorithm for taking a global snapshot of a running distributed system of actors [32].

12.5.3 Coordination

Dynamic, virtual organization of agents can be accomplished by using coordination mechanisms to express a wide variety of interactions. Coordination is a key design concern for a multi-agent system. Since each problem-solving agent possesses only incomplete information which represents a local view of the overall system, and limited computational power, it must coordinate with other agents to achieve globally coherent and efficient solutions. Coordination can be viewed from three different perspectives: the information content, the exercise of control, and the coordination mechanisms [26]. The information used for coordination can be data, new facts discovered, partial solution/plan, preferences, or constraints. What one would like to develop are reusable abstractions for coordination which allow agents to play a richer variety of roles.

As a gross simplification, temporal coordination can be seen as an abstraction of synchronization, the problem of determining *when* actions take place rather than *what* individual actors do. Hence, coordination constraints are an abstraction of synchronization constraints, constraints on the order of actions.

It turns out that two types of synchronizations are often useful. The first type imposes precedence constraints on otherwise asynchronous events at different actors, and the other requires such events to be atomic (loosely speaking, to co-occur). By providing a language abstraction, called *synchronizer* to express these two types of constraints, we are able to show that the task of distributed programming may be further simplified [10]. Because synchronizers may be superimposed, and may be dynamically added or removed, implementing such a system efficiently proves to be a fairly challenging but is nevertheless feasible. The following example is due to Frolund [11].

Example 12.7 Coordinating Robots
Consider two coordinating robots. Each robot has an arm and a hand, and it can grab a widget with its hand, and lift and move it using its arm.

A single robot can be modeled as a part-whole hierarchy where the robot object serves as an interface between a user and the robot components. When told to move a widget from point p_1 to point p_2, the interface tells the arm the hand to p_1, tells the hand to grab the object, tells the arm to move the hand to p_2, and finally tells the hand to release the object. At the completion of any request, the component (hand or arm) informs the robot object about the completion. For instance, the hand would send the message `releaseDone`

Here, we'll consider the case where two robots are to cooperate in moving widgets. The top level object is a logical robot `composed` that serves as an interface for the composed physical robots. These composed robots are allowed to share a widget that is at a position reachable to both. A request may involve movement of a single robot or it may need cross-robot movement. To service a latter type of request to move a widget from p_1 to p_2, the interface robot would tell robot closer to p_1 (the `passer`) to move it from p_1 to $p_s h$, the shared position, and next tell the other robot (the `receiver`) to move it from $p_s h$ to p_2. The `passer` would in turn communicate with `passerHand` and `passerArm` and so on. Depending on the physical details of the environment, cross-robot movement may have integrity requirements, such as:

- *Totality:* The top level message must send a move message to both or neither of the robots. If only one robot can be dispatched, the widget may get "stuck" at the shared position, preventing cross-robot movement involving other widgets.
- *Collision avoidance:* At most one widget may occupy the shared position at any time.
- *Sequencing:* During a cross-robot movement, the first robot must release the widget before the second robot grabs it.

A synchronizer to coordinate cross-robot movement would have to represent each of these requirements. The totality and collision-avoidance requirement are satisfied by putting an atomicity constraint, that requires `move` requests to both robots to be dispatched at the same time. The sequencing requirement is satisfied by disabling `receiver`'s hand from grabbing the widget while `passer` is active, and by installing triggers that would alternate the value of `passerActive` between `T` and `nil`, as it is dispatched `move` and `releaseDone` messages (by `composed` and `passerHand`, respectively).

```
(defSynch robots (passer receiver receiverHand start end shared)
  (let ((passerActive nil))
       (atomic (request-when (passer.move from to)
                    (and (= from start) (= to shared)))
               (request-when (receiver.move from to)
                    (and (= from shared) (= to end))))
       (disable (request-when receiverHand.grab passerActive)
       (trigger (-> (request-when (passer.move from to)
                                  (and (= from start)
                                       (= to shared)))
                    (setf passerActive T)))
                (-> (request-when passer.releaseDone T)
                    (setf passerActive nil)))))
```

The `robots` synchronizer template is instantiated by the top-level object `composed` for each cross-robot movement. □

Synchronizers can be very effective in enforcing system level coordination requirement such as the need to avoid redundant work. Note that in a multi-agent system, multiple agents serving the same interest often end up performing the same execution sequences without knowing about each other. At the system level, such redundant activity could be avoided by using appropriate synchronization constraints to disable requests for an activity following the first one.

Example 12.8 Real-Time Constraints
RTsynchronizers [24] offer one way of implementing real-time constraints using an abstraction similar to that for the declarative coordination constraints discussed earlier. RTsynchronizers are objects that enforce real-time constraints by constraining whether or not messages of a certain type can be delivered to an actor at a certain point in time.

Consider a variation of the Producer/Consumer problem where the produced object must lie in the buffer for a certain amount of time before being removed

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

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

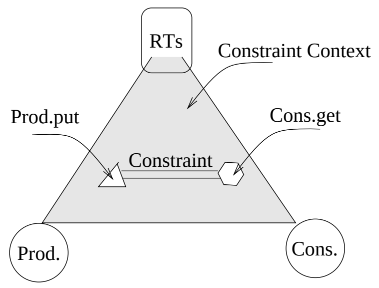

Figure 12.5 Producer/Consumer with Time-Bounded Buffer

by the consumer. A *Time Constrained Producer/Consumer* problem can be implemented by writing writing the code for the usual Producer Consumer problem without explicitly considering the time constraint (Figure 12.5). Then, separately an RTsynchronizer can be declared with the time constraint that would prevent the Consumer's `get` request to be delivered until the required amount of time has elapsed. The declaration are translated into the correct scheduling of actors, if such a translation is feasible. □

12.5.4 Naming and Groups

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

Groups of agents are an important unit of representation; for example, in defining protocols we can assign roles to a group of agents rather than an individual agent. Moreover, it is often necessary to communicate with agents whose address is 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 object request broker architecture perform a similar function.

The ActorSpace model allows an abstract specification of a group of actors [7]. An actorspace associates an actor with specific attributes; the sender of a message specifies a destination pattern which is pattern-matched against the attributes of actors in the actorspace. The model may also be seen as providing a distributed version of the *blackboard*[8] system for broadcast communication. A simple analogy

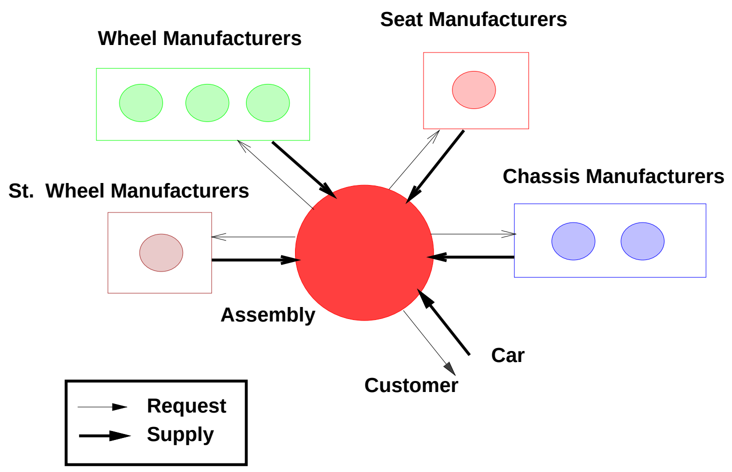

Figure 12.6 A car assembly factory. The assembly sends requests to actorspaces whose membership may dynamically change.

with set theory illustrates the difference between naming in actors and actorspaces. A set may be defined by enumerating its elements, or by specifying a characteristic function which defines a subset in a domain. The first method is analogous to actor communication (where an explicit collection of mail addresses of actors must be specified), whereas the second method corresponds to actorspace communication. Of course, in conventional mathematics the two ways of characterizing sets are equivalent since the properties of mathematical objects are static; by contrast, actors may dynamically change their attributes. Actorspace provides a transparent way of managing groups of actors. It generalizes the notion of ports in process calculi, where object identity is also not uniquely defined, but pattern are degenerate.

Figure 12.6 shows a simple example of an actorspace. A car assembly requires certain types of parts which may be available through different vendors, sets that may themselves be changing over time. Which vendor fills a request may not be germane to the assembly process. Such requests may be mediated through an actorspace. Finally, meta-level operations may be associated with an actorspace. For example, an actorspace manager may transparently schedule requests to ensure load balancing.

12.6 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 [12], which executed in a distributed memory multiprocessing environment. Based on the experience of this research, Les Gasser [13] 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 [35] has been developed at MCC.

Genesereth [14] defines an agent as an entity that is able to communicate correctly in an agent communication language, thereby emphasizing the expressiveness of such a language. Programs may be converted into software agents by rewriting them so that they have the needed communication ability, or by employing transducers or wrappers to achieve such functionality. Facilitators keeping track of capabilities of agents implement a federated system of communication providing a pattern-based message sending facility.

The Knowledge Query and Manipulation Language (KQML) [9, 21], described in detail in Chapter 2, is a message-handling protocol that aims to provide an effective platform for agent communication by addressing fundamental components of (i) a common language, (ii) a common understanding of exchange knowledge, and (iii) an ability to exchange the two. KQML messages communicate an attribute called *attitude* along with the message content. The language primitives, called *performatives*, define actions permissible to agents in communication. There are special agents called *facilitators* that provide support in identifying agents and services, brokering agreements, etc.

The term Agent Oriented Programming has been coined by Shoham [27] 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. The latter idea is derived from speech act literature (e.g.[25]) which categorizes speech in similar ways. Each agent runs a loop in which it first reads the current message, updating its mental state, and then executes the commitments for the current time. Munindar Singh [28] has developed a theoretical framework for reasoning about intentions, know-how and communications.

A multi-level architecture for Multi-Agent Systems is described by Werner [33] where a meta-architecture is defined to formalize users', programmers' or designers' interactions with an open system. Michael Kolb's CooL (Cooperation Language) [19] 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. We give two examples to illustrate such agents.

In Chapter 14, Java has been discussed in some detail. 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 [20] developed at OSL attempts to put a discipline for system development in Java.

The Mobile Agent Facility Specification by the Object Management Group [17] 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 [34] 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 [16] 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.

12.7 Conclusions

The ability to coordinate the behavior of agents in agent ensembles is a key challenge for Distributed AI. We are just beginning to understand the concept of agent and the requirements for supporting their execution. A platform for supporting multi-agent ensembles needs to provide scalable mechanisms for safe and efficient execution over open networks of computers. No such architectural platform currently exists today.

We have presented some basic notions that are necessary to support programming agents for DAI, but it is by no means the complete picture. In particular, the underlying platform must control ways in which resources are accessed and managed. The chapter has described how resource allocation policies may be represented at the agent level. Research on techniques for resource allocation continues and will be able to borrow from previous work in subject areas as diverse as operating systems and economics.

Our current understanding of agent semantics is still primitive. For example, there is no well developed equational theory of agents. Because such a theory would allow rigorous reasoning about the behavior of agents, it is very important to the problem of security. Specifically, nodes must be protected against malicious or buggy agents. One idea is that a host could verify the relevant properties of an agent before admitting the host in a less protected mode. Because finding a proof of a program is computationally very expensive (it can be intractable), agents could carry proofs of their programs that the hosts check. Checking an existing proof is computationally much less expensive.

Another approach to security is to *sandbox* the agents. This technique, partially employed by the programming language Java, physically separates the space occupied by an important piece of code (such as that for an agent), to prevent it from affecting the node's operation in any undesirable way. However, because Java's sandboxing model does not limit the physical or logical resources consumed by imported code, it is insufficient for preventing deleterious agents. A third possibility is authentication of agents. Hosts would allow access to agents based on prior knowledge or by checking certification provided by trusted registries.

From a different perspective, because agents can spawn other agents, multi-agent systems must also be able to control the activity of ensembles of agents. The behavior of an individual member of an agent ensemble may be quite reasonable, but the behavior of a group of agents can be chaotic. We have a number of examples where the outcome of collections of autonomous processes result in this kind of phenomena. Consider two of them. A ferry sunk as all the passengers rushed to one side in response to a perceived emergency. A power outage in a small area caused a cascading outage. Economic models of control, such as those in markets, may be one approach here. However, for reasons that are apparently not entirely understood, the short term behavior of markets with human players can itself be quite chaotic.

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.

12.8 Exercises

1. *[Level 1]* Security is an important concern in multi-agent systems. Systems may be threatened by harmful activities of individual agents or by ensemble of agents affecting system performance by their collective activity. One such concern, resource consumptive behavior, has been discussed at some length in this chapter. Describe other specific ways in which security is threatened in such systems by collective behavior of agent ensembles.
2. *[Level 1]* Download an Actor system and use it to implement a parallel search of a distributed n-ary tree. You can find a Java-based Actor system at `<http://osl.cs.uiuc.edu>`.
3. *[Level 1]* Describe at least three different schemes for implementing actors in Java. Discuss the advantages and disadvantages of the design decisions in each scheme.
4. *[Level 2]* Implement an interpreter for an actor language and develop a single processor simulation of actors for executing programs written in this language.
5. *[Level 2]* Extend the semantics developed in this chapter to incorporate a yellow pages service. Specifically, provide a way of representing and maintaining ActorSpaces, and write new transition rules needed to express communication in a system employing ActorSpaces.
6. *[Level 3]* In an actual implementation of an agent architecture, it may be unreasonable to assume that the resources needed to complete a task can be predicted reliably, requiring a more complex mechanism by which agents may request more resources from, or return unused resources, to a *sponsor*. These sponsors may be created by client agents as managers of resources available to a task. Extend the semantics described in this chapter to incorporate a reasonable scheme employing such sponsors. Note that potential frequency of sponsor-agent communication may preclude having remote sponsors; similarly, a naive scheme may result in the sponsors becoming a bottle-neck.
7. *[Level 3]* Agents in a multi-agent system may be organized in different ways. An example would be a group of agents learning to solve an optimization problem using the genetic algorithm. Implement such a system and study its ensemble level behavior.
8. *[Level 3]* Consider several representative types of organizations of agents and study potentially chaotic behaviors that may result at the level of ensembles. Specifically, analyze systems organized as *markets* and *firms*. What types of desirable emergent behaviors can you expect to result from such organizations.
9. *[Level 4]* Design and implement an agent architecture. Document the assumptions you make about how agents and agent ensembles would use the architecture.

12.9 References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.
2. G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Proceedings of the 3rd IFIP Working Conference on Dependable Computing for Critical Applications*, September 1992.
3. G. Agha, C. Houck, and R. Panwar. Distributed execution of actor systems. In D. Gelernter, T. Gross, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, number 589, pages 1–17. Springer-Verlag, 1992.
4. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1996. to appear.
5. Gul Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.
6. Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.
7. C. J. Callsen and G. A. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.
8. 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.
9. Tim Finin, Yannis Labrou, and James Mayfield. Kqml as an agent communication language. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 291–316. MIT Press, 1997.
10. S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proceedings of ECOOP 1993*, volume 707. Springer Verlag, 1993.
11. Svend Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press, 1996.
12. 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.
13. 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.
14. Michael R. Genesereth. An agent-based framework for interoperability. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 317–346. MIT Press, 1997.
15. Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents for mobile computing. Technical Report PCS-TR96-285, Department of Computer Science, Dartmouth College, Hanover, NH 03755, May 1996.
16. 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.
17. Crystaliz Inc., General Magic Inc., GMD FOKUS, and IBM Corporation. Mobile

Agent Facility Specification. Technical report, Object Management Group, June 1997.

18. W. Kim. *Thal: An Actor System for Efficient and Scalable Concurrent Computing.* PhD thesis, University of Illinois at Urbana-Champaign, 1997.

19. 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.

20. Open Systems Laboratory. The actor foundry: A java-based actor programming environment. *Available for download at http://www-osl.cs.uiuc.edu/ ~astley/foundry.html.*

21. Yannis Labrou and Tim Finin. A proposal for a new kqml specification. Technical Report CS-97-03, University of Maryland Baltimore County, February 1997.

22. P. Maes. Computational reflection. Technical Report 87-2, Vrije University. Artificial Intelligence Laboratory, 1987.

23. G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

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

25. J. Searle. *Speech Acts.* Cambridge University Press, Cambridge, UK, 1969.

26. M.J. Shaw and M.S. Fox. Distributed artificial intelligence for group decision support. *Decision Support Systems*, 9:349–367, 1993.

27. Yoav Shoham. An overview of agent-oriented programming. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 271–290. MIT Press, 1997.

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

29. 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.

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

31. 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.

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

33. 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.

34. James E. White. Mobile agents. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 437–472. MIT Press, 1997.

35. D. Woelk, M. Huhns, and C. Tomlinson. InfoSleuth agents: The next generation of active objects. *Object Magazine*, July/August 1995.