The Spider Model of Agents

F. Y. Huang D. B. Skillicorn
Department of Computing and Information Science
Queen's University, Kingston, Canada
{huang,skill}@cs.queensu.ca

June 2001
External Technical Report
ISSN-0836-0227-
2001-447

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared June 18, 2001
Copyright ©F.Y. Huang and D.B. Skillicorn

Abstract

We take the position that large-scale distributed systems are better understood, at all levels, when locality is taken into account. When communication and mobility are clearly separated, it is easier to design, understand, and implement goal-directed agent programs. We present the *Spider model* of agents to validate our position. Systems contain two kinds of entities: spiders which represent service providers, and arms, which represent goal-directed agents. Communication, however, takes place only between an arm and the spider at which it is currently located. We present both a formal description of the model using the ambient calculus, and a Java-based implementation.
Keywords: agent models, locality, reasoning, Java.

The Spider Model of Agents

F.Y. Huang and D.B. Skillicorn
{huang,skill}@cs.queensu.ca

Abstract: We take the position that large-scale distributed systems are better understood, at all levels, when locality is taken into account. When communication and mobility are clearly separated, it is easier to design, understand, and implement goal-directed agent programs. We present the *Spider model* of agents to validate our position. Systems contain two kinds of entities: spiders which represent service providers, and arms, which represent goal-directed agents. Communication, however, takes place only between an arm and the spider at which it is currently located. We present both a formal description of the model using the ambient calculus, and a Java-based implementation.

1 Motivation

We present a distributed agent system, called the *Spider Agent Model*, which is designed to be structurally transparent, both to reasoning and to agent task design, and efficient. The model distinguishes two kinds of entities: spiders, which rarely move and play the role of service providers, and arms, which play the role of agents and are fully mobile.

The design of the spider agent system is motivated by two aspects of existing distributed agent systems which we consider as weaknesses. These are:

1. Most existing systems allow goals to be achieved *both* by communication and by mobility. When there is only one way to accomplish any goal, it is easier to design the system appropriately, and it is profitable to devote resources to optimize the implementation of the only one possible solution. When there are multiple ways to accomplish a goal, it is hard for users to understand the system, it is hard to choose the best strategy for implementing a particular action, and it is hard to know where best to spend resources to improve performance. A good example is the world wide web which presents, to the user, an illusion of mobility but where all activity is actually implemented by communication. It is hard for a novice user to become more sophisticated; for example, the role of proxies is difficult to comprehend if the user's mental model assumes browser mobility.

   The spider agent system implements information sharing at a distance by mobility, and information sharing locally by communication. These two aspects are clearly different within the system.

2. Existing systems confuse two concepts that we will call *virtual mobility* and *physical mobility*. Any distributed system of non-trivial complexity uses virtual names for its objects and a mapping mechanism that associates physical names with them. This mapping need not remain fixed over time.

   When an object can move physically, but is still accessible using the same virtual name, then we say that it has physical mobility. Cell phones are physically mobile:

their virtual name is their associated phone number, while their physical name depends on the cell they are in at any given moment.

When an object's virtual name can also change, then we say that it has virtual mobility. For example, when a person moves from one company to another, she exhibits virtual mobility, since all forms of access have changed.

Humans handle physical mobility without difficulty, since it does not require changes to our mental maps – only the virtual name needs to be remembered. Virtual mobility is more difficult – human organizations are *not* constructed to use it, except on very slow time scales. Building artificial systems that are different from human systems is a recipe for opacity at best, and unusability at worst.

Many agent systems confuse these two kinds of mobility, and sometimes go to great pains to implement virtual mobility, which we regard as misguided. For example, ambients do not distinguish between the two kinds of mobility [3]. One ambient can be 'absorbed' by another, which is not a behaviour with many direct analogues in the real world.

Of course, the distinction is one of degree not of quality, since it depends on how much work is required to manage redirections. Virtual mobility could be concealed by a further layer of indirection. In the end, the distinction is really whether a name can be resolved to a location in a small constant number of steps (physical mobility) or more (virtual mobility).

In the spider agent model, the service objects (spiders) are physically mobile but not virtually mobile – they maintain the same virtual name throughout.

We show that imposing these limitations on the spider agent system makes it simple to understand and expressive. The extra structure also makes it easier to reason about the behaviour of agents within the system.

The spider agent system is an open architecture in which agents are light-weighted and flexible. Spiders are able to offer arbitrary services, and arms (agents) may contain code that interacts with some or all of the services it encounters.

The description of the spider agent system in this paper and the implementation details represent preliminary work. Important open questions remain to be answered.

Section 2 describes some related work. Section 3 describes the spider model in detail. Section 4 introduces two styles of reasoning about programs written using the spider model. The ambient style is discussed in detail. Section 5 describes the prototype implementation.

2 Related Work

Wooldridge and Jennings [8] give four properties for agents: *autonomy*, agents can act without intervention from outside; *reactivity*, agents can perceive their environment and act in response; *proactivity*, agents are goal-directed; and *social ability*, agents are able to coordinate their strategies with other entities. In a distributed system, one of the actions that an agent can use is to move from one to another processor, making it a *mobile agent*.

There are many models for agents and agent systems. Useful overviews can be found in [1] and [4]. We highlight three types:

1. Agent systems with CORBA-like goals, that is the ability to assemble components that are physically distributed to make useful wholes. Examples include: Voyager from Objectspace (www.objectspace.com), and Concordia from Mitsubishi Electric (www.meitca.com).

2. Aglets, from IBM Japan, which might best be considered an agent system infrastructure or standard [5].

3. Ambients [3], a general approach to computation in space with a firm semantics.

There are also well-developed systems for reasoning about agents, many based on extensions of standard ways of reasoning in distributed systems. The Ambient model represents a development of ideas from process calculi, and particularly the π-calculus. Such systems are extremely general and powerful, and are often motivated by assuming an environment in which the objects and their actions are constantly changing. This is only a realistic assumption if the environment is considered to include (a substantial part of) the whole system. If communication and mobility are decoupled, then agent actions are associated with a particular location in space. The environment in which they interact is relatively static, altered only by arrivals and departures of other agents, and therefore easier to understand and reason about. An extension of the ambient calculus, called Safe Ambients[6], defines co-action capabilities corresponding to ambient actions for synchronization and interference control of concurrent ambients.

Here is a simple example of ambient interaction synchronized by co-actions. Ambient k in ambient m is equipped with capability to enter ambient n by prior arrangement. It moves from m to n then continues with process P_k. Ambients m and n have their own processes P_m and P_n as well:

$$\begin{array}{ll} & m\,[\,\overline{out\,m}.\,P_m \mid k[out\,m.\,in\,n.\,P_k]\,] \mid n\,[\,\overline{in\,n}.\,P_n\,] \\ \rightarrow & m\,[\,P_m\,] \mid k\,[\,in\,n.\,P_k\,] \mid n\,[\,\overline{in\,n}.\,P_n\,] \\ \rightarrow & m\,[\,P_m\,] \mid n\,[\,P_n \mid k[P_k]\,] \end{array}$$

Other approaches to reasoning about agents use temporal [7] or modal approaches to modelling what agents know or believe.

3 The Spider Model

The spider model contains two entities:

1. Spiders, which play the role of service-providing objects. Spiders have unique permanent names, and occur in hierarchies, called *spider domains*. Spider names, therefore, have the form *spider_name@domain_name*.

   Spiders all provide certain basic services related to arm admission, resource allocation, movement and termination. Spiders are typed, and all spiders of a given type provide

a known set of services associated publicly with that type. Individual spiders are also free to provide specific services.

2. Arms, which play the role of mobile agents. An arm is created attached to a particular spider (its *home spider* with which it remains associated in a special way as long as it remains an arm). Arms are free to move to other spider domains (if admission policies permit) where they may make use of the services of the (local) spiders in these domains. Arms may return to their home spider, they may choose to die in any spider domain, or they may choose to *settle* inside a spider domain and become a new (subsidiary) spider (if policies permit).

   Arms do not have accessible names.

There is a clear separation of mobility and communication. Communication is always local, between an arm and its host spider in the domain where the arm is currently located. Mobility is therefore necessary whenever the information required to meet goals cannot be obtained locally. Note also that communication is always asymmetric, between entities of different kinds, so that agent programs with communication deadlocks cannot be written.

Spiders may also move, but their mobility is incidental to their function. For example, a spider may reside on a laptop. When the laptop is disconnected from the Internet at one location and reconnected at another, the spider has moved, but this has no effect on its behaviour as a spider, nor on any agents currently located in its spider domain. Arms seeking to move to the relocated spider must follow a different path to find it, but this redirection is really a function of the underlying network.

Allowing arms to communicate only through intermediary actions of spiders imposes structure on the patterns of actions of agents. On the one hand, this is a significant limitation, since emergent complex behaviour based on the interaction of many simple agents is harder to express. On the other hand, it does permit computations to exploit locality – and agent need only be prepared for what it might encounter within each spider domain at a time, not for everything it might encounter in the entire system. As the "entire system" increasingly becomes an entity that spans the globe, this is a major saving in complexity, both intellectual during design of the agent, and performance by reducing the amount of code an agent must carry against contingencies. Enforcing communication via spiders also means that resources can be held in the static pieces of the system (the spiders) rather than carried around in the mobile pieces (the arms).

Separating communication and mobility enforces a data-centric view of a distributed computation, in which code moves towards data rather than the other way around. This makes better use of network bandwidth, which may be important if parts of the network are wireless.

3.1 Spiders

A spider domain consists of a hierarchy of spiders, each of which is interacting with a set of arms. These arms are of two kinds: the spider's own arms which it has created, typically in response to a request from a user (located 'at' this spider domain) that requires accessing

remote data; and arms from other spiders that are currently located at this spider ('just visiting').

Spiders admit visiting arms based on their security policies. An arm admitted into a spider domain must initially ask for one of three things: to die, to be moved to some other spider domain, or to be given a set of resources. Whether, and how much of these resources are granted by the spider depends on its local policies, but it is important that all types of resources are requested and granted atomically to prevent resource deadlocks within spiders. In the prototype, the only resource considered is computation cycles.

A spider provides a resource-bounded playground for each arm in which it may consume the allocated resources in any way it wishes. However, this typically involves interactions with the spider invoking any of the services that this spider provides.

Spiders may know the names of other spiders, and may reveal these to arms as a service. Requests to move are at the instigation of arms, which must know the name of the spider to which they wish to move. However, wild cards in names are possible, in which case the current spider is free to move the arm to any matching destination spider. Including wild cards enables arms to access services without having to know the names of individual spiders.

3.2 Arms

As described in the previous section, an arm arriving in a spider domain typically begins by asking for a grant of resources. After this, it may interact with the spider using any of the following actions:

- Reconfiguration. Spiders guarantee to provide arm code for certain generic parts of arm structure, and may also contain certain kinds of type-specific arm code. Hence an arm only needs to carry (a) enough code to gain access to a spider domain, and (b) code specific to its mission, since it can pick up other code inside spider domains. This can be implemented using Java's mechanisms to include code from packages at different locations.

- Standard services. These include:

  - Die. Remove this agent from existence inside the spider domain.
  - Move. This requires a spider name to be given as an argument. The arm is repackaged for movement and sent to the given destination.
  - Settle. The arm is given standard spider code and becomes a descendant of the current spider in the hierarchy, if this is permitted. If not, control returns to the arm to take appropriate action.

- Specialized spider services. Spiders of the same class offer services standard to that class. An individual spider may also offer particular services. The interface for services is generic, indexed by an unbounded service number. Information about which spiders offer which services is ordinary data and accessible in ordinary ways.

The necessary parts of the spider model have been kept as small as possible. Useful systems require more than this basic set of services. For example, a spider name service,

search engine spiders, and spiders that maintain a persistent public storage area (e.g. in the style of the MARS project [2]) are all likely to be common extensions. Notice that many standard web-based activities are naturally implementable using the spider model, with the important difference that browsing, search, and so on actually use mobility. Applications such as cooperative search require spiders with public persistent storage. Cooperating arms can use such storage to communicate with each other – but note that communication is spider-mediated and asynchronous making it much harder to create trivial deadlocks.

4 Reasoning about the Spider Model

Spider programs (that is, actions taken by spiders at the request of an external user) can be reasoned about in two styles.

The spider model is a subset of the ambient model, and so reasoning in the ambient style is possible. For example, confidence in the implementation can be increased by ensuring a match between implemented actions and rewrite rules within the ambient representation.

Reasoning can also be done in a modal style based on what is known by each spider and arm. We can, for example, define predicates such as *knowsname(a,s')*, *pathbetween(s, s')*, and *admissible(a, s')* where a is an arm, and s and s' are spiders. These predicates can be automatically verified from the code of each object. From these, we can define a relation, *canmoveto(s, a, s')*, which is reflexive and transitive, and so for which closures can be computed. These closures allow reasoning about reachability.

If security is a concern, then we can make an assumption such as "an arm can discover anything known to a spider it visits", and extend the reachability relation above to a *canfindoutabout* relation which describes the way in which information flows in the system as a whole.

4.1 A Language for Reasoning

Based on the description of the Spider Model in section 3, two entity types of the model are defined in Tables 1 and 2.

In Table 1 it is supposed that $q \in QoR$ where QoR is a set of values on which a partial order, addition, subtraction and $\underline{0}$ are well defined. Definition of QoR is application-dependent. There may be both unrecoverable resources like CPU cycles and recoverable resources like memory, and different applications may focus on different types. We will consider only unrecoverable resources here.

rm_a in Table 2 is the resource manager for arm a created by its host spider, responsible for resource approval, deduction of consumed resource, and termination of arm a when it runs out of resource.

There are four types of services provided by each spider, corresponding to the four basic types of service request arm actions. Other services can be provided and requested through a general parametric interface *service*, which is linked to specific services that differ from spider to spider. A spider can contain child spiders as well as arms. Arms can only be active within a spider and can communicate only with its host spider. The interactions between arms and spiders are represented by the reduction rules in Table 3. (The convention of

Table 1: Definition of Arm

a, b	arm names
A, B :=	arm processes
$\mathbf{0}$	inactive process
$A \mid B$	parallel processes
$E.A$	arm action
E :=	atomic action of an arm
C	arm's internal computation action
R :=	arm's service request action on host spider
$moveto(s)$	ask to be moved to spider s
die	ask to be killed
$reqres(q)$	resource request
$settle(s)$	ask to be converted into spider s
$getserv(p)$	request other services
q	a value representing some quantity of resource
p	a set of values containing service id and parameters

Table 2: Definition of Spider

s, t	spider names
S, T :=	spider processes
$(\nu a)S$	restriction of arm name
$\mathbf{0}$	inactive process
$S \mid T$	parallel processes
$s[T]$	child spider
$a[A]$	residing arm
V :=	service responding to arms' requests
$moveArm$	move an arm to a specific spider
$receiver$	authenticate incoming arm
$rm_a(q)$	resource manager for arm a
$kill$	kill an arm
$settleArm$	convert an arm into specific spider
$service$	parametric interface for other local services, returning different result types for different service id specified in parameter lists
r	a set of values representing particular result returned from $service$ for particular set of parameters
x	variable

Table 3: Reduction Rules

Mobility:

$s[\, a[moveto(t).\, A] \mid moveArm \mid rm_a(q)\,] \mid t[receiver]$
$\rightarrow s[moveArm] \mid t[\, a[A] \mid receiver \mid rm_a(q_{min})\,]$ (R-mov)

Resource authorization:

$s[\, a[reqres(q_1).\, A] \mid rm_a(q)\,] \rightarrow s[\, a[A] \mid rm_a(q + q_{q_1} - q_{rr})\,]$ (R-auth)

Resource exhaust:

$s[\, a[A] \mid kill \mid rm_a(\underline{0})\,] \rightarrow s[kill]$ (R-exht)

Arm internal computation:

$s[\, a[C.\, A] \mid rm_a(q)\,] \rightarrow s[\, a[A] \mid rm_a(q - q_C)\,]$ (R-intl)

General service request:

$s[\, a[getserv(p).\, A] \mid service \mid rm_a(q)\,]$
$\rightarrow s[\, a[A\{x \leftarrow r_p\}] \mid service \mid rm_a(q - q_{gs(p)})\,]$ (R-serv)

Termination:

$s[\, a[die] \mid kill \mid rm_a(q)\,] \rightarrow s[kill]$ (R-die)

Settlement:

$s[\, a[settle(t)] \mid settleArm \mid rm_a(q)\,] \rightarrow s[\, t[T] \mid settleArm\,]$ (R-sett)

substitution notation is adopted: $A\{x \leftarrow value\}$ means substitution of all free occurrences of variable x in process A by $value$.)

q_{min} in R-mov is an initial amount of resource that is just enough for the incoming arm to request further resource or to move to somewhere else if local resources are not granted. Value q_{q_1} in R-auth is the granted resource based on the requested amount q_1, the spider's resource situation, and the resource management policy. Note that the request action itself consumes some resource q_{rr} ($q_{rr} < q_{min}$). Value r_p in R-serv is the returned result from the requested service (suppose that A needs the result for variable x). Services related to interface $service$ can be identified and invoked by their IDs or names included in the parameter list.

For all the reduction rules except R-exht and R-die, we assume that resource q is sufficient for the requested service. If not, the corresponding reduction cannot be completed and the arm will be killed in-process by R-exht.

A simple but typical application of the reduction rules is an arm a moving from spider s to t then asking for resource q_{set} to settle down as spider s_a with services V_1 and V_2. (Assume that the security and resource policies permit it to do that.)

$s[\, a\,[\, moveto(t).\, reqres(q_{set}).\, settle(s_a)\,]\,|\, moveArm\,|\, rm_a(q)\,]\,|\, t\,[\, receiver\,|\, settleArm\,]$
$\rightarrow \quad s\,[\, moveArm\,]\,|\, t[\, a\,[\, reqres(q_{set}).\, settle(s_a)\,]\,|\, receiver\,|\, rm_a(q_{min})\,|\, settleArm\,]$
$\rightarrow \quad s\,[\, moveArm\,]\,|\, t[\, a\,[\, settle(s_a)\,]\,|\, receiver\,|\, rm_a(q_{min} + q_{set} - q_{rr})\,|\, settleArm\,]$
$\rightarrow \quad s\,[\, moveArm\,]\,|\, t[\, s_a\,[\, V_1\,|\, V_2\,]\,|\, receiver\,|\, settleArm\,]$

4.2 Encoding in the Ambient Calculus

A formal language to express and reason about any applications of the Spider Model requires a lot more logic, such as structural congruences and related reduction rules, to be defined

for semantic soundness. Another way to work is to express it using another sound language. It is obvious that the Spider Model is conceptually a subset of ambient model. In Table 1 and Table 2 it is clear that they are just specific ambients. By encoding the reduction rules in Table 3 using ambient calculus, we show that interactions between arms and spiders are just ambient actions.

For example, before considering resource consumption, the arm's mobility request can be encoded as:

$$\begin{aligned} moveto(t).A = \quad & mt[\, out\, a.\, in\, ma.\, \overline{open\, mt}.\, \langle\, armID, attr_t, in\, a\, \rangle\,] \\ & \mid \overline{out\, a}.\, \overline{in\, a}.\, open\, admit.\, (x).\, \overline{out\, a}.\, out\, s.\, x.\, A \end{aligned}$$

where $attr_t$ is a set of spider attributes that can be used to match spider t. The spider's mobility service can be encoded as:

$$\begin{aligned} moveArm = \quad & !\,(\, moveArm' \mid open\, ma\,) \\ moveArm' = \quad & ma[\, \overline{in\, ma}.\, open\, mt.\, (y_1, y_2, y_3). \\ & \qquad (mapt(y_2).\, \overline{open\, ma}.\, \overline{out\, s}.\, \overline{in\, s} \mid (y_4).\, ra(y_1, y_3, y_4)\,)\,] \end{aligned}$$

$mapt(y) \rightarrow^* \langle in\ t_y \rangle$ (if spider t_y is known)

$ra(y_1, y_3, y_4) = ra[\, out\, s.\, y_4.\, in\, rv.\, \overline{open\, ra}.\, \langle y_1,\ in\, s.y_3 \rangle\,]$

$$\begin{aligned} receiver = \quad & !\,(\, receiver' \mid open\, rv\,) \\ receiver' = \quad & rv[\, \overline{in\, rv}.\, open\, ra.\, (z_1, z_3).\, (\, veri(z_1).\, \overline{open\, rv}.\, \overline{out\, t} \mid admit_t(z3)\,)\,] \end{aligned}$$

$veri(z) \rightarrow^* \varepsilon$ (if arm id z passes the authentication)

$admit_t(z_3) = admit\,[\, out\, t.\, z_3.\, \overline{open\, admit}.\, \langle in\, t \rangle\,]$

Including resources consumption makes the encoding is more complicated, and only unrecoverable resource that can be represented by numerals in ambient calculus [3, p18] have been considered.

5 Implementation

A prototype of the Spider Model has been implemented using Java, with arm mobility based on Java object serialization. A general arm holds an unique ID and a pointer to the current host spider, and has methods representing basic actions described in Table 1 for programmers to invoke. These methods are light-weight because they just call corresponding spider services.

```
public class Arm extends Thread implements java.io.Serializable
{
   private transient Spider hostSpider;//the current host spider
   private AgentID myID;               //arm's ID for authentication
   ......
   protected boolean moveMeTo(SpiderAddress spiderAddress) {...}
   protected void die() {...}
   protected Resource requestResource(Resource resource) {...}
   protected boolean settle(Vector serviceNames) {...}
   protected Result getServ(String serviceName, Vector parameters){...}
   ......
   public AgentID showID() {...}
   public void run() {...}
}
```

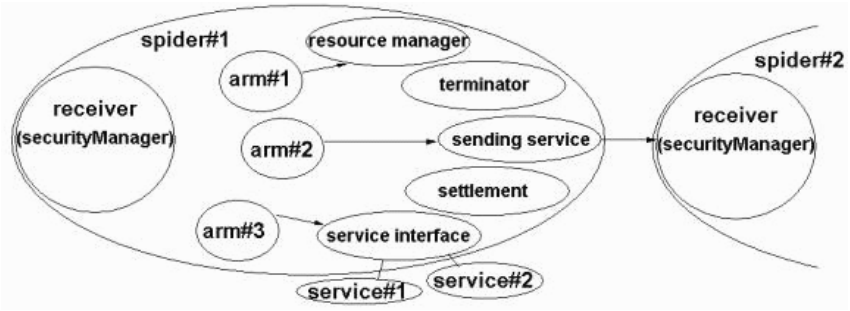

Figure 1: Architecture of A Spider

Based on the description in Section 3 and Table 2, the pattern of the architecture of a spider is also clear, as illustrated in Figure 1.

First of all, each spider executes an *armReceiver* on a particular port number. The *armReceiver* is responsible for receiving incoming arm code, authenticating for admission, recreating the admitted arm from code and allocating initial amount of resource for it. If the arm is rejected, the sender spider is notified. Complicated systems can have a separate *security manager* to maintain a sophisticated security policy.

Mobility service consists of receiving and sending services. When an arm requests move service, the host spider will contact the receiver of the destination spider at the particular port.

Another important part of a spider is the *resource manager* which maintains a resource policy, deals with resource requests from residing arms, and monitors resource consumption. Note that although many other models and systems use the phrase *resource manager*, they usually mean a data storage/allocation manager, a completely different concept.

The spider services are invoked by arms through corresponding public method calls. These are the only way for arms to interact with their environment. Spiders keep local pointers to residing arms private.

```
public class Spider extends Thread
{
   private SpiderAddress hostSpider;//the location of parent spider
   private SpiderAddress myLocation;//the location of this spider
   private Resource resource;       //resource granted by parent
   private Hashtable myArms;        //the residing arms
   private Hashtable childSpiders;  //child spiders
   private Vector services;         //names of provided services

   private ArmReceiver armReceiver = new ArmReceiver();
   private ResourceManager resMgr = new ResourceManager(this, myArms);
   ......
   public Spider(...)
   {  ......
      resMgr.start();              //start the resource manager
      armReceiver.start();         //launch the arm receiver
   }
```

```
   ......
   public boolean moveArmTo(Arm arm, SpiderAddress destination){...}
   public boolean kill(Arm arm) {...}
   public Resource grantResource(Arm arm, Resource requestAmt) {...}
   public boolean settleArm(Arm arm, Vector servNames) {...}
   public Result service(Arm arm, String servName, Vector params){...}
}
```

To show the simplicity of programming using the spider model, the code of a specific arm that explores a set of spiders is shown below. It is given an initial itinerary. When it arrives at a new spider, it first spends some time doing local work (`doMyTask`), and then moves to the next spider currently at the front of its itinerary.

```
public class ExplorerArm extends Arm
{  public static final int REQUIRED_CPUTIME = 5000; //in ms
   public static final int NUMBER_OF_STOPS = 10;

   private int requiredCPUTime = REQUIRED_CPUTIME;
   private Vector itinerary = new Vector(NUMBER_OF_STOPS);

   public ExplorerArm(Spider hostSpider, AgentID myID)
   {  ......      //given the itinerary
   }

   public void run()
   {  if (itinerary.isEmpty())
      {  System.out.println(this + " finished task, dying...");
         die();
      }

      if (! showID().getOwner().equals(getHostLocation()))
      {  Resource grantedRes=requestResource(new Resource(requiredCPUTime));
         if (grantedRes.compareTo(new Resource(requiredCPUTime)) >= 0)
            doMyTask();
      }

      String destName = null;
      while (!itinerary.isEmpty())
      {  destName = (String) itinerary.firstElement();
         itinerary.remove(destName);
         moveMeTo(new SpiderAddress(destName, Spider.DEFAULT_PORT));
         System.out.println("No spider available at host "+destName);
      }

      //no more spider host available, go home.
      if (!moveMeTo(showID().getOwner()))
      {  System.out.println(this + " becomes homeless, dying...");
```

```
            die();
        }
    }

    protected void doMyTask()
    {  ......
    }
}
```

This code is very stylized. A much more abstract language, in which agent actions were described at the level of "move there", "search for this" can be straightforwardly mapped (compiled) to such code. Hence users can describe the actions taken by arms without needing to be explicit about the details of how they are accomplished.

6 Conclusions

Our position is that large-scale distributed systems are better understood, at all levels, when locality is taken into account. It is more natural, more efficient, and easier to reason when the concepts of communication and mobility are clearly separated and clearly visible in the model.

To support this position, we have designed and implemented the spider agent model. The distinguishing features of the model are:

- Two kinds of entities: spiders, which represent service providers, and arms, which represent goal-fulfilling distributed computations.
- Insistence that communication can only take place locally (that is, within a spider domain), so that there is only one way to acquire remote information – by moving to the location where it exists. Hence, mobility is not an extra, optional feature, but a necessity.
- Because of restrictions on form, there is typically only one way to achieve any particular goal. This helps with design clarity, and also directs attention to those system aspects that most repay optimization.
- Because of the restrictions on form, reasoning about program behaviour is simplified.

The spider agent system implementation is preliminary, and space has prevented a full discussion of its design, and well as more detailed examples of reasoning about programs.

References

[1] P S K Booker, R K Granger, E J Guest, S A Norton, J E Price, and H Glaser. Software agents and their use in mobile computing. Technical Report DSSE-TR-99-5, Declarative Systems and Software Engineering Group, University of Southampton, February 1999.

[2] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A Programmable Coordination Architecture for Mobile Agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[3] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378, pages 140–155. Springer-Verlag, Berlin, Germany, 1998.

[4] Neeran M. Karnik and Anand R. Tripathi. Design issues in mobile-agent programming systems. *IEEE Concurrency*, 6(3):52–61, 1998.

[5] Danny B. Lange. *Java Aglet Application Programming Interface White Paper*. IBM Tokyo Research Lab, February 1997. Online: http://www.trl.ibm.com/aglets/JAAPI-whitepaper.html.

[6] Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *Symposium on Principles of Programming Languages*, pages 352–364, 2000.

[7] M. J. Wooldridge. *The Logical Modelling of Computational Multi-Agent Systems*. PhD thesis, University of Manchester, Manchester, UK, 1992.

[8] M.J. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.