MOBILE CODE, DISTRIBUTED COMPUTING AND AGENTS

J Waldo

Rapporteur: T Rischbeck

MOBILE CODE, DISTRIBUTED COMPUTING, AND AGENTS

Jim Waldo
Sun Microsystems, Inc. and Harvard University
1 Network Drive
Burlington, MA 01803 USA
jim.waldo@sun.com

Introduction

Traditional approaches to distributed computing, at least those that are based on the model of Remote Procedure Call (RPC)[1], have been built on a number of assumptions about the environment in which they are to be deployed. These assumptions have led to the set of abstractions that make up the computational model of such systems, paying attention to some of the aspects of the system while ignoring others. Such abstractions are needed if we are to think about general systems at all. But occasionally it is useful to check the assumptions that underlie such a computational model to make sure that the environment that shaped those assumptions has not changed so much as to make the assumptions no longer useful.

It was just such an examination that led to the construction of the Jini™ network technology system. The Jini system rejects a number of the fundamental assumptions that have been at the foundation of previous RPC-based systems. In particular, the Jini system assumes a single underlying implementation language for distributed objects (the Java™ language) and (via the Java platform) the ability to move objects (including their implementation classes) from one part of the distributed system to another.

Because of this use of mobile objects, the Jini system has sometimes been categorized as a mobile agent system. However, such a categorization is based on an over-simplified view of mobile agent systems. In fact, there are characteristics of mobile agent systems that clearly differentiate such systems from the Jini technology. Unfortunately, these characteristics also make such systems highly unlikely to actually be useful, at least in the opinion of the author.

Ironically, a number of the problems that are inherent in mobile agent systems can be addressed if those systems are implemented in the kind of environment offered by the Jini system. By making slight changes to the characterization of mobile object systems they can be fit into the Jini environment in a simple, consistent fashion that will allow new approaches to most (but not all) of the basic problems that beset mobile agent systems.

In what follows, we will briefly describe some of the features of the Jini networking system, and attempt to show how the assumptions of this system are different from those that have been used in the construction of other distributed system frameworks. We will then turn to a characterization of mobile agent systems, showing both how such systems are intrinsically different from the Jini system, and what some of the basic problems of

---

1. The classic definition of RPC is in Birrell and Nelson[1]. However, the approach has a long history, including the Network Computing System[2], DCOM[3], and the OMG CORBA[4] systems.

such systems are. Finally, we will discuss how agent systems could use the Jini system approach to solve some of these intrinsic problems.

The Jini Networking System

A full discussion of the Jini networking system[5] is beyond the scope of this paper. However, a characterization of some of the major features of the system will help in what follows.

The Jini networking system is a distributed infrastructure built around the Java programming language and environment. The basic communication model is based on the semantic model of the Java Remote Method Invocation system, in which objects in one Java virtual machine communicate with objects in another Java virtual machine by receiving a proxy object which implements the same interface as the remote object This proxy object deals with all communication details between the two processes. The proxy object may introduce new code into the process to which it is moved. This is possible because Java bytecodes are portable, and is safe because of the built-in verification and security of the Java environment. It should be emphasized that the Jini system uses the RMI *semantic* model; it does not require that the communication over the wire be done using the RMI implementation contained in the Java platform. Since that communication takes place between the remote implementation and the proxy object that came from that implementation, it can be done using RMI or by using other communication mechanisms such as CORBA, XML-based messaging, or a custom communication protocol.

To this underlying communication model the Jini system adds some basic infrastructure and parts of a programming model. The infrastructure provides a mechanism by which clients and services can join into the Jini network, while the programming constructs encapsulate mechanism that allow reliable distributed systems to be built.

The infrastructure centers around the *Jini Lookup Service*. This service provides a place for providers of a service to advertise their availability, and for clients desiring a service to find what services are available. Bootstrapping is accomplished using the *Jini discovery protocol*, by which an entity (either hardware or software) can find a Jini lookup service when it first connects to the network. The discovery protocol is the specification of a multicast packet which, when received by a Jini lookup service, will result in the lookup service sending back to the broadcaster a Java object that implements the lookup service interface[2]. This object can be loaded into the process of the sender, which can then register itself with the lookup service if it wishes to provide (one or more) services and to query the lookup service for other service providers. Note that a Jini participant can be both a service and a client of other services.

Services describe themselves by the Java language interfaces that they implement. When a client wants to find a service, the client will request an object that implements a particular Java interface. Any object that is an `instanceof` the requested type can be

2. The discovery protocol is specialized for each kind of network on which Jini systems run. The one discussed is the discovery protocol for tcp/ip networks. Other discovery protocols, with other mechanisms, will need to be defined for networks with different properties (such as, for example, Bluetooth or Firewire networks). However, the result of the discovery protocol will be the same on all networks--the entity doing discovery will receive an object (or objects) that implement the Jini lookup service interface.

returned to the requestor; in particular, objects that implement a subtype of the requested type, or which implement a set of types that includes the requested type, may be returned.

The interfaces that encapsulate the programming model center around three sorts of object interactions. A set of interfaces define the notion of an event notification, allowing an object to export the ability to other objects to register for notification when some conceptual event occurs in the first object. A set of transaction interfaces define a two-phase commit protocol that can be used by objects to coordinate activities within the objects. Finally, a leasing interface defines a model of time-based resource allocation, in which a resource is granted for a renewable period of time, determined by a single-round negotiation between the resource requestor and the resource granter. Renewal of a lease can be requested by the holder of the lease, but if the lease is not renewed before the expiration time of the lease, both the grantor of the lease and the holder of the lease can assume that the resource that was leased is to be freed for others to use.

The infrastructure within the Jini system uses the Jini programming model. In particular, the lookup service allows registration of interest in a number of kinds of events, allowing participants in a Jini network to track changes to the set of services available to them via the lookup service. Registration of a service within a lookup service is also leased, so that if a service does not renew the lease the service is deleted from the set of services available in the lookup service. This has the result of insuring that the lookup service does not contain service advertisements for services that no longer exist for any period of time longer than the maximum lease period granted by the lookup service, as such services will not renew their leases.

From this brief description, it should be clear that the Jini system is tied to the Java programming language and platform. Java provides a mechanism that allows mobile objects, including their code, to be safely and efficiently moved from a service to a client of that service. The Java type system is used to identify services, and the polymorphic nature of that type system allows requests for a service to be treated as requests for something that implements at least a certain type, although it may offer more. However, it should be noted that the requirement for the Java language and platform is only at the level of the network; non-Java objects can be used to implement a Java network object that can live within the Jini world.

Mobile Code and Mobile Agents

The use of mobile objects with mobile code in the Jini system has led some to characterize the system as a mobile agent system. The reasoning seems to be something along the lines of

1) Mobile agent systems use mobile objects
2) The Jini system uses mobile objects
3) Therefore, the Jini system is an mobile agent system

The problem with this argument, of course, is that it is an instance of an invalid argument form. Just because both Jini and mobile agent systems use mobile objects, it does not follow that the first is an instance of the second. The only way in which an argument like this could establish the conclusion would be if the first premise were a much stronger proposition of the form

1') Any system that uses mobile objects is a mobile agent system

If 1') were the first premise of the argument, then it would logically follow from this premise and the fact that the Jini system uses mobile objects that Jini was a mobile agent system. But while the argument would be valid, it is not at all clear that it would be sound, because the new premise does not seem to be true. The proposition stated in 1') says that the use of mobile objects is a sufficient condition to make a system be a mobile agent system. While it seems obvious that moving objects (in a loose sense) is a necessary condition for a mobile agent system (since agents can be equated with objects), and while it might also be true that most of the systems that make use of mobile objects have, in fact, been mobile agent systems, the movement of objects does not seem to be the only thing that characterizes a mobile agent system.

Let us consider a couple of examples of mobile agent systems, of the sort that are often given in discussions of mobile agents. The first of these has to do with shopping over the Internet. Rather than requiring that I go out and search the net for the best deal on, say, an airline ticket from Boston to London, I should be able to send an agent out into the network to do the work for me. This agent will move from Internet site to Internet site, finding out what the times, prices, and availability of flights are for the times in which I am interested. It might simply gather the information and return to me, but a more common example is one in which the agent, upon finding the best deal, books my travel (using my credit card numbers, which it keeps securely) and returns to tell me that all has been done, and when I need to have my bags packed.

A second example shows the power of mobile agents when combined with the kinds of proximity networks that promise to soon be available. Upon entry into a cab, my personal agent, which resides on my personal digital assistant (PDA) device, joins into the wireless proximity network within the cab. Consulting my schedule (also on my PDA), my agent determines that I am going to the airport, and moves to the location of the cab's agent. My agent tells the cab's agent where I'm going. The two agents then negotiate a price (and perhaps a route), my agent pays (again using my credit card information), and my agent returns to my PDA, entering the transaction into my financial statement.

A final example returns us to the Internet, or at least the World Wide Web. In preparing a paper, I discover that I have neglected to do the scholarly research expected by the journal to which I am hoping to submit the work. I don't want to use the simple search engines, since I know that the number of false positive returns will far outweigh the number of good returns when I make my query. So I send an agent out on the Web after telling it the subject of my paper, and my agent jumps around the web and returns to me the next morning, with the six most relevant papers to my subject.

I don't know if these are real examples of what mobile agent systems are supposed to do. They are certainly the kind of examples that one sees in the popular press, and I have been given exactly these examples in conversation with researchers in the field when I ask for an example of what a mobile agent system could do. However, I have also discovered that there is considerable disagreement in the mobile agent community about just what makes up a mobile agent system and what such systems are supposed to do. Examples like those I have given may well constitute a straw man in which no one

believes. However, such examples are the common view of what mobile agent systems are about.

These examples (and others that I have heard) all have a number of characteristics, which I have taken to be the characteristics of a mobile agent system. From these examples, I take the characteristics of a mobile agent system to include

1) the use of *active* objects, that is, objects that move from place to place and, when they arrive at a place, are able to obtain a thread of control automatically;
2) the agent will *interact* with the environment in the various places it finds itself. Simply moving to a new machine is not enough; the agent moves to that machine and then finds out information or performs actions at that machine;
3) the agent will obtain information or perform actions on *behalf* of the person who sent the agent out on the network;
4) the agent, after doing its work, will *return* to the person who sent it out on the network and *report* the results of the work; and
5) the agent is capable of *making decisions* on behalf of the person who sent it out on the network (or, more generally, on behalf of the person whose agent it is).

These characteristics seem to be the ones that make the above examples both interesting and clear examples of mobile agents. They provide a way of marking a clear distinction between mobile agents and other distributed systems or simple web crawling. There may be other characteristics that I have missed, or some that I have overstated or mis-stated. For example, it has sometimes been claimed that characterization 5) is actually what distinguishes systems of *intelligent* mobile agents from mere systems of mobile agents. Such distinctions may be needed sometime, but not here. These are the characteristics of mobile agents that I will discuss in what follows.

The first point to make is that, given this characterization of mobile agent systems, the Jini system is not a mobile agent system. While objects can be mobile in the Jini system, such mobile objects are not active in the sense of automatically getting a thread of control. A mobile object in the Jini system must get a thread of control explicitly from the process to which it has been moved, either explicitly (if the object is a subclass of `Runnable`) or implicitly (when some method on the object is called). Mobile objects in the Jini system may interact with the Java environment on the machine to which they are sent, but it is much more likely that they will interact with other Jini services that reside in the network. There is no requirement that a mobile object have the identity of some principle who sent the object, nor that the object return to any point of origin. Finally, there is no requirement that mobile objects in the Jini system be capable of making any decisions on behalf of their sender; indeed, one of the most common forms of mobile object in the Jini system simply forwards calls on to the service it represents.

In fact, it is just these characteristics which, I will now argue, have doomed mobile agent systems to be the disreputable ghetto of distributed computing. The very characteristics that make a system a mobile agent system also make such systems difficult or impossible to deploy in the real world.

The first characteristic, that of using active objects, means that mobile agent systems blur the line between the resources that the owner of a particular machine controls and the

resources that are controlled by the agents that arrive at that machine. If an agent that arrives on a machine automatically gets a thread of control on that machine, then the allocation of computational resource is automatic. This makes little or no difference in cases when the number of agents arriving is small and the machines are not resource constrained. But this approach does not scale, either to resource limited machines or to large numbers of agents.

Large numbers of agents can cause an agent storm, in which even a generally capable machine can be brought to a standstill by the chance occurrence of a large number of mobile agents arriving on the machine at the same time. As the number of agents roaming the network increases, the chance of being hit by such a storm increases. And as the network moves to being composed of smaller machines (like cell phones and PDAs) the number of agents that can, in effect, cause a denial of service attack decreases.

The second characteristic, that the agent will interact with the environment on the machine to which it moves, assumes a homogeneity of destination machines which is hard to take seriously in a large-scale network. When an agent arrives at a new destination, how does it know what to interact with? How does it do so? Does it read the information that it needs from a file, and if so where in the file system does that file reside? Or does it contact some service on the machine, in which case it needs to know how to contact that service in a way that works on all of the possible destinations to which it will be sent.

The third characteristic, that the agent is able to obtain information or perform actions on behalf of the person who sent the agent, assumes a form of distributed authentication far more sophisticated that is generally available. For the agent to act on my behalf, I need to be able to delegate some or all of my identity to that agent. While security systems that have delegation have been proposed, few have been successfully implemented, and the problems (such as being able to cancel a delegation or insuring that delegations are bound in some fashion) are not easily solved.

Further, the security needed for a mobile agent system requires properties that are not required in general distributed system security. Since the agent is being moved onto a host, it needs to be able to be authenticated. But when does that authentication take place? Since the agent is active, by the time the recipient knows that the agent is there, the agent is running, perhaps able to do damage to the host machine. Further, how does the agent (since it is often carrying sensitive information, such as credit card numbers and authorizations) establish trust in the machine on which it is going to run? Since the agent is living in the address space of a process that was given to it by the machine on which it is running, the agent can not be sure that it has any secrets from the host machine.

The fourth characteristic, that the agent will return to its machine of origin after it has completed its work, seems reasonable until we consider the fact that networks, and the machines on them, fail. Suppose that the machine on which an agent was running crashes. How long does the entity that sent out the agent wait for that agent's return before sending out another? Suppose that the network link from the current machine to the home machine is down. How long does an agent wait until it gives up the return trip? The failure models for agent systems do not seem well specified, yet any distributed

system which does not have a failure model that deals with these kinds of failures will not enable the building of reliable applications.

Finally, the last characteristic of agent systems, that they be able to make decisions on behalf of the owner of the agent, the seems to require that we solve at least part of the traditional problems of artificial intelligence. I have not seen many computer programs that are very good at making decisions, and certainly not any that I would be willing to have make decisions on my behalf, whether those decisions have to do with travel plans or the relevance of papers. I am not saying that such programs are impossible (I remain a skeptical agnostic on this subject). However, those that seem to come the closest to having this capability can all be characterized as large programs. Yet mobile agents, if they are to be useful, need to be small entities that can easily move over the network without swamping the machines to which they move. Even in moments when I feel most optimistic about the prospects of artificial intelligence, I don't see such decisions being made by small, mobile objects that can run anywhere.

For these reasons, I have never taken the claims of the mobile agents community terribly seriously. The Jini system was certainly not designed as a mobile agent system itself; it was much more modest in its goals. The Jini system was never intended to be either a mobile agent system itself, or a system that could be used to construct such systems. The irony, at this point, is that a number of the problems that have plagued mobile agent systems can be solved in rather straightforward ways when those systems are based on the Jini technology and are run in a Jini-enabled network.

New Directions in Mobile Agents

The first problem with mobile agent system is that they employ active objects, and therefore have automatic access to the resources of the machines to which the agent travels. A solution to this problem would be to move the agents from one machine to another as passive objects, but move them to a Jini service (advertised in the lookup service by its type) which would then be responsible for the granting, at the appropriate time and at the appropriate priority, of a thread of control to that agent.

The design behind such a scheme is fairly straightforward. The agent host service would have an interface that would include a `receive` method, which would take an object of an `Agent` type (perhaps simply defined as a Java `Runnable`). Upon receipt of such an object, the agent host object could start up the agent, or queue the agent for later activation, depending on the current load and the policy established by the owner of the machine onto which the agent had moved.

Building such an agent host is simple. In fact, it is given as an example for the Java Remote Method Invocation system[6], under the guise of a compute server. The service itself is about 30 lines of code. turning that service into a Jini service (which requires finding a lookup service, registering with that service, and renewing leases on that registration) requires about 100 more lines of code.

It could be objected that this is, in effect, what current active object agent systems do under the covers, and so the stated approach is no different from current practice. In one sense, this objection is correct; from a high level view the description of what happens is no different. However, when agents are activated by some service rather than by a part of

the underlying system, control over the policies used by the service are left with the owner of the service. If I run an agent host service on my machine, I can control the conditions under which an agent is activated. All that is required of the agent host is that it implement a known interface. How that interface is implemented is up to me, not a part of a general system. If this is a more general system activity, then I have given up control (at least in the agent systems I have seen). I could adopt a policy that agents are to be activated immediately upon receipt, in which case I would be subject to the kinds of agent storms discussed earlier. But I could also adopt a very different policy (likely after experiencing my first major service degradation) which would ration the resources given over to the arriving agents. The main point is that it is up to me, the owner of the service, and not up to the agent system.

An agent system that is running in a Jini-enabled network also has a general technique for finding appropriate information when an agent moves from place to place. Since one of the major goals of Jini is to allow service discovery over the network by programs, the mechanisms of lookup by Java type can be used by agents when they reach a new location to find the services needed (or discover that such services are not available at the location). Once an agent gets a thread of control, the agent can find the local lookup service, and then query the lookup service for objects that present the programmatic interface that the agent needs for its work.

The major difference between this approach and those that are based on names (such as file systems) or descriptions (like directory services) is that the Jini style of lookup is designed for use by programs rather than people. Naming systems and directory systems rely on strings, with the semantic information contained in the meanings of those strings. This is a good system for people, since people are very good at mapping strings to the meanings represented by those strings. But programs are very bad at this kind of understanding; those that are even capable of basic understanding are far too large to be moved around in the fashion of mobile agents.

Programs, however, are very good at understanding types; something that people tend to do far less well. Indeed, even programs that attempt to find information using naming or directory systems translate the name or description into a type (i.e., the methods that can be used to access the information in the named or described entity). In a sense, using the type based matching of the Jini system simply allows the agent to eliminate one level of indirection, mapping directly from the methods that the agent will use to access information to the entities that can provide that information, rather than having to first find an entity by name or description and then hope that the entity will support the method calls that are contained in the agent.

If the Java type system were not polymorphic, this would be the only important difference between matching by type and matching by string. This is no small difference, as it means that there is only one level of uniform convention that needs to be agreed to throughout the system rather than three (the names, the types, and the mappings from names to types). But since Java is a polymorphic language, the difference is more significant than just a reduction in the number of required agreements.

When an agent (or any other service client) asks the lookup service for something, it asks for something that is at least of a particular type, although it may be more. Objects that

implement a subtype of the requested type can be returned, as can objects that implement interfaces other than those that were requested. Since the Java language supports reflection, the client can discover that more has been returned than was requested, and (if the client knows how) make use of the other methods supported by the object.

For an agent system, this means that agents can make use of services that provide more or additional information than the agent absolutely requires, or that have evolved (by extension) since the agent was first written or deployed. By using a type-based lookup, the agent gets the advantage of an environment in which it can find what it needs in a simpler fashion, as well as allowing the environment in which the agent works to change in ways that will not effect the functioning of the agent itself. This is the major functional difference between the string-based lookups used in naming and directory services and the type-based lookup used in Jini: in the former, only exact matches are possible, while in the latter you can match on something which is at least what you are looking for, but may be much more.

Agent systems built within a Jini framework can also make use of the Jini/Java RMI security model, which is currently being defined through the Java Community Process. The draft specification for this security model already includes mechanisms for mutual authentication of client and service and a time-based delegation model. More important for use within the agent community, the model provides a mechanism to allow authentication to occur before objects are reconstructed. This is important, as such reconstruction generally requires the running of some of the code associated with the object (the constructor). Unless trust is established prior to the running of any code, the system would be open to various forms of attack.

This model is layered on top of the existing Java security models, which (while not perfect) give a model for insuring that bad behavior on the part of downloaded code will be detected prior to the running of that code, and provides a mechanism to limit the access of any downloaded code to resources on the destination machine. The security model tends to err on the side of caution (as is proper for a security model), and use by the agent community would be a valuable source of input into the continued evolution of the security mechanisms for the language and platform.

Finally, the implementation of an agent system within the Jini technology model would begin to provide a failure model that might be of use to the agent community. In particular, the notion within Jini of a distributed object (that is, one that exists on multiple machines at the same time) and the Jini programming model of a lease can provide solutions that can be used in the common cases of failure within an agent system.

Jini encourages the construction of objects that, at least conceptually, live on multiple machines. The application of this model to agent systems would entail the agent itself being such a distributed object, with a part that moved from place to place and a part that stayed on the machine of origin. Partial results that are obtained by the roving part of the agent can be sent back to the more sedentary fraction of the object as they are obtained. Such a mechanism would minimize the effects of a machine on which the roaming part of the agent currently resides failing. Further, since the sedentary fraction of the agent could be given access to stable storage (something far less likely to be given to the roaming part

of the agent) the temporary results of the roaming agent that were sent home could be stored in a way that allowed survival of a crash on the home machine.

This model of the agent as distributed object does not help with the problem of what to do when contact between the agent and the home machine has ceased, either because of a network failure or because one of the components is on a machine that has failed. However, the leasing model in the Jini system can provide a mechanism for the two parts of the distributed agent to coordinate in such circumstances. If the roaming and sedentary part of the agent have agreed upon a time interval for communication, then when that time expires without communication having successfully taken place, each part of the agent can take action appropriate (which can be defined by the individual implementation of the agent). For example, the roaming part of the agent, upon failure to make contact with the sedentary part of itself during the time of the lease, could simply cause itself to cease to exist, knowing that the sedentary part of the agent will assume that this is what it has done under this circumstance. The sedentary part of the agent, when the lease expires, can know that it will receive no more communication from the roaming part of itself, and can either re-initiate a roaming agent (sent out to do only the work that remains) or decide that it will make due with the partial results that it has already obtained.

There are interesting interactions between these approaches which will only be understood when systems have been built trying to use them. For example, since this model makes the roaming part of an agent inactive until granted a thread of control by the agent host, the lease interval between the sedentary part of an agent and the roaming part of an agent will need to be long enough to allow for the roaming part of spend some period of time waiting for a thread of control from an agent service. Perhaps it is possible for the roaming part of an agent to hand off lease renewal to the agent host until the agent becomes active, although such a handoff might require that the agent or some part of it be active prior to full activation, which has certain security implications. Building an agent system inside of the Jini framework is still a challenge, but some of the challenges at least seem to have promising avenues for solution.

Unfortunately, the Jini technology does nothing to help with the ability of agents to make decisions on behalf of the sender of the agent. This is a fundamental problem in agent systems, and there is nothing in the Jini technology that will help in this. Indeed, much of the design center of the Jini technology was predicated on the supposition that software is not capable of making intelligent decisions. The system designers felt that the problems of distributed computing were hard enough that they did not need to add the problems of artificial intelligence into the requirements set.

Bibliography

[1] Birrell, A.D. and B.J. Nelson. *Implementing Remote Procedure Calls*, **ACM Transactions on Computer Systems** 2 (1978).

[2] Zahn, L., T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, and G. Wyant, **Network Computing Architecture**, Prentice Hall (1990).

[3] Microsoft Corporation, **Object Linking And Embedding Programmers Reference, Version 1**. Microsoft Press (1992).

[4] The Object Management Group, *Common Object Request Broker: Architecture and Specification*. OMG Document Number 91.12.1 (1991).

[5] Arnold, K., B. O'Sullivan, R. Scheifler, J. Waldo, and A. Wollrath, **The Jini Specification**, Addison Wesley (1999)

[6] Campione, M. and K. Wallrath, **The Java Tutorial Continued**, Addison Wesley (1999).

DISCUSSION

Lecture One

Rapporteur: T Rischbeck

Dr Waldo's first talk focussed on the advantages the Jini technology offers for distributed programming. As implementation details are hidden by a Java language layer, this means an "end of the wire protocol hierarchy".

Mr Peine started the discussion by questioning, whether this comes at the expense of the "new tyranny of having to use the Java programming language". However, as Dr Waldo pointed out, the language approach taken in Jini is more flexible compared to the standard protocol approach to distributed programming. In Jini, a small Java bytecode for the stub is everything required to get a client to "talk" with the server object. As Java bytecode may be produced by a variety of languages (Dr Waldo: "even a Cobol compiler exists"), one is not tied to the Java language. Dr Waldo emphasized, that compared to competing technologies, like Microsoft's ActiveX, security is an inherent feature of the Java platform, and mentioned as example the JVM's bytecode verification unit.

Professor Marneffe asked for details on how Java's versioning system works -- which he mentioned as having prime importance for Jini. Dr Waldo explained that Jini supports a subtyping mechanism and gave as example the implementation of a printing service. This service would implement a printing interface, exhibiting all methods necessary for client-control. A stub object, implementing the same interface, is downloaded from a code repository to clients. Clients then call methods on this stub object to utilise the printing service in a network-transparent fashion. If at a later stage a colour-printing service is introduced, this is described as a sub-interface of the printing interface. Whereas new clients could be programmed to use the new interface, old Clients could still use the old interface without breaking.

Dr Waldo went on, that the real problem comes from environment evolution, e.g. the move from Java 1.1 to Java 1.2. The solution is to only allow interfaces as method arguments. More complete code could be downloaded on demand. Efficiency problems in such a setting might be resolved using caching.

Professor Sloman brought up the issue, whether bytecode verification can determine object termination properties. This he mentioned would be relevant to distributed garbage collection techniques in order to decide whether objects are required for later stages of program execution or not. Dr Waldo replied that such a feature is not available and was doubtful whether it could be implemented.

Professor Andre finally concluded the session by referring to intellectual property issues and their aptitude to capture technologies like Jini. Dr Waldo agreed, that this topic needs consideration, but also refered to previous discussions.

DISCUSSION

Lecture Two

Rapporteur: T Rischbeck

Professor Randel initiated the discussion with the observation, that semantics are not clearly described by the interface definitions in Jini. As an example, he brought up the cowboy/draw anecdote: Two interfaces are both implementing a draw method. But as he explained, the meaning can be quite ambigious: Is the destination object a pencil, supposed to draw a line on canvas, or is it a cowboy who draws his gun? Dr Waldo said that the only way around is by comments accompanying the interface methods.

The next contribution came from Professor Sloman. He mentioned the similarity between Carl Hewitt's Actors and nowadays Agents. An inherited requirement, he said, is the need to include aspects of negotiation and contracts as key notions for any realisitic mobility model. Dr Waldo replied, that Jini in fact establishes this kinds of contracts.

Professor Shrivastava was interested in the scalability of the Jini messaging systems. As he said, most business computation systems rely on asynchronous communication for scalability reasons -- those are not provided in Jini. Dr Waldo countered, that truly asynchronous communication is not desirable, because the sender cannot establish knowledge about the receiver's reception of the message (e.g., email). Another problem is failure handling, which deserves serious consideration in distributed systems. In summary, Dr Waldo expressed the opinion, that aysnchronous communication is great (for performance/scalability reasons), unless the network fails, which should be taken into consideration for distributed systems.

Dr Waldo listed three alternatives to asynchronous messaging:

1. Linda (e.g., shared tuple space, compare JavaSpaces).
2. Caller threads spawned for every call, responsible for collecting the result.
3. Call and acknowledgement; but after sending the acknowledgement, the callee may proceed asynchronously; Dr Waldo emphasised the difference between asynchronous calls and asynchronous computation as obvious in this technique.

Professor Marneffe suggested the possibility of extending the TCP/IP protocol to find out about receiver status in asynchronous communications. Dr Waldo replied with a classification of four possible failure states:

1. message never received (by callee)
2. machine crash before the transaction
3. machine crash after the tranasction
4. reply lost

A caller cannot know when to send the message again. All solutions rely on a reestablishment of the communication at a later time. The only viable solution is transactions, but they are expensive to implement. Dr Waldo suggested the seminal paper on "end-to-end communcation" for more information on this topic.

Dr Waldo mentioned that the Jini communication model is synchronous, in response to a question by Dr Ezilchelvan. This was followed by a remark from Professor Randell, that there might be a tension between a realistic and a "copable" fault model. However, as Dr Waldo stated, just any fault model is required for agent programmin now. Specifically, researchers should focus on the question, whether failure semantics in agent system extend what is already known from distributed systems.

Resource requirements might be a necessary feature to be described in an agent language, Dr Thomsen remarked. Dr Waldo agreed that it is important to worry about memory and processor resources an agent demands. An abstraction over resources is required.

Mr Peine pointed out, that the issues Jini tackles have all been dealt with previously. E.g., service discovery, resource management, etc. Those are not specific to mobile agents. Dr Waldo made the statement, that the agent community and the distributed community differ in the way they are dealing with these issues. Often, he said, those are just ignored in an agent context.

Finally, Mr Cunningham raised the opinion, that he would favour an independent standards body for the advancement of Agent technology, rather than relying on an industrial and commercially oriented player, like Sun Microsystems. He mentioned the Foundation for Intelligent Physical Agents (FIPA, www.fipa.org) as such a standards body, taking input from many sources, rather than following the "one does it all" approach which Sun takes.