Paradigma: Agent Implementation through Jini

Ronald Ashri and Michael Luck
Department of Computer Science
University of Warwick
Coventry, CV4 7AL, UK
Email: {R.Ashri,mikeluck}@dcs.warwick.ac.uk

Abstract

One of the key problems of recent years has been the divide between theoretical work in agent-based systems and its practical complement which have, to a large extent, developed along different paths. The Paradigma implementation framework has been designed with the aim of narrowing this gap. It relies on an extensive formal agent framework implemented using recent advances in Java technology. Specifically, Paradigma uses Jini connectivity technology to enable the creation of on-line communities in support of the development of agent-based systems.

1 Introduction

In a networked environment that is highly interconnected, interdependent and heterogeneous, we are faced with an explosion of information and available services that are increasingly hard to manage. Agent-based systems can provide solutions to these problems as a consequence of their dynamics of social interaction; communication and cooperation can be used to effectively model problem domains through the interaction of agents. Before the agent paradigm experiences widespread use, however, there are many issues that need to be resolved [5]. Perhaps most important, however, is the very urgent need for agent development methodologies and agent frameworks to enable developers to implement agent-based systems effectively and quickly. The development of such systems can also provide much needed experience to aid in answering other questions.

In this paper we describe an effort to address this need in providing an implementation framework for developing agent-based systems using recent advances in Java technology. Moreover, the work also addresses the theory-practice divide in using Luck and d'Inverno's [6] formal agent framework as the base for the implementation. Specifically, we describe the Paradigma implementation platform which, using Jini connectivity technology, allows for the creation of on-line communities. The aim of Paradigma is to let the developer concentrate on the definition of agents and other entities through standard techniques, and minimise the effort on their implementation, by taking care of issues of behaviour control as well as discovery, communication and interaction with other entities.

The paper begins by briefly reviewing Luck and d'Inverno's formal agent framework as a base for understanding the aims of the implementation environment, the design principles and underlying technologies which are discussed in the subsequent section. Then the details of how objects and agents are created, and how they can be instantiated within Paradigma, are explained. Before presenting conclusions covering further work and other possibilities opened up for agent-based systems by the technologies used, some consideration is given to how communication between autonomous agents is achieved.

2 A Formal Agent Framework

The notion of agents is to a large extent vague and uncertain, and inherently allows for a variety of interpretations. Many alternative definitions have been presented in the literature, usually excluding other approaches. In an attempt to ground their own work in particular agent concepts and to allow it to move forward and tackle other more important issues, Luck and d'Inverno describe an inclusive formal framework that precisely delineates the notions of agents and autonomy, but without rejecting alternative forms. In essence, their proposal is for a four-tiered hierarchy containing entities, objects, agents and autonomous agents [6].

In this view, entities comprise a set of attributes, which are simply describable features of the world. Objects are entities with a set of capabilities, which are actions an object can perform that cause an effect on the state of the environment. Agents are objects with a set of goals defined as sets of desirable attributes and, finally, autonomous agents are those agents able to generate their own goals through mo-

tivations that drive them. Motivations can be thought of as preferences or desires of an autonomous agent that lead it to produce goals to satisfy those desires.

Thus, a clear distinction is made between the notions of agents and autonomous agents. Agenthood is ascribed to any entity in the world that acts in order to satisfy some goal, and when those goals are self-generated the agent is autonomous. This simple framework has also been further refined to accommodate more sophisticated analyses of agent interaction, by introducing additional definitions of neutral objects as those objects in the world that are not agents and server agents as those agents in the world that are not autonomous [3]. Neutral objects give rise to server agents when they adopt goals. Once those goals are achieved, or pursuing those goals is no longer feasible, the server agent reverts back to a neutral object. This is a significant characteristic of the framework on which we will focus later in this paper when we explain how the framework is implemented as a real set of APIs. In any case, however, we will not discuss the agent framework here, but leave the reader to explore previous work [3, 6].

In order to make these abstract notions clearer, however, suppose that you want your personal agent (PA), with motivations such as minimising on-line connection time, saving money, and providing comfort, to find the best train ticket to visit London for you. Given this task, it generates goals according to its motivations, including that of getting a list of all travel agents providing tickets to London, and tries to achieve it by locating an object in the environment with the capability of providing such lists. The PA engages the object and thus instantiates a server agent from it with the goal of providing the required information. Now, the actions of the object required to accomplish this may be nothing more than a multicast announcement requesting active travel agents to register. Having received the list, the PA attempts to contact the travel agent using some form of communication language. Having acquired the required information (limiting the number of calls so as to satisfy the motivation of minimising on-line time, and getting a cheap ticket according to the other motivations), the PA reports the results and waits for further instructions.

This example illustrates some key points. First, it is important to be able to define entities that do not necessarily exhibit traditional standard agent qualities, but which can be regarded as agents because it is useful for us to do so (for example, when given tasks to accomplish, or goals to satisfy). Second, interactions between entities can take different forms. They could be direct engagements of an object or a cooperation between two autonomous agents (as discussed in [7]). Finally, motivations are required to drive autonomy. An agent needs to have some innate driving forces that will define its behaviour and allow it to generate or choose the next goal. In the example, the autonomous agent has the motivation of minimising on-line time so it could prefer the goal of downloading all the information locally before processing it.

3 Paradigma Agent Implementation

Design Principles As stated earlier, the aim of Paradigma is to provide an agent implementation environment based on the agent framework described above. In a sense, Paradigma can be seen as implementing the agent framework, and both the framework and the environment serve as infrastructure in their respective domains in which agent development (both theoretical and practical) can take place. Based in this way on a principled theoretical model, Paradigma was also guided by certain design principles that structure its general development. Technologies such as Jini and XML have been used, enabling the framework to gain from their evolution as well as aiding in its understanding by other developers. In addition, we opted for providing a grounded model that can be easily changed and expanded, thus allowing us to adapt the model as our understanding of the agent paradigm evolves, as opposed to an all inclusive solution. Finally, agent implementation has been separated from agent description, through the use of XML, in order to allow for the construction of different agent types.

Java for Agent Systems As argued by Kinny et al. [4], while object oriented (OO) programming languages cannot provide a direct solution to agent development, features such as abstraction, inheritance and modularity make it easier to manage increasingly more complex systems. The increased complexity of agents and their behaviour means that despite the power of OO techniques, they cannot directly by mapped to agent design. However, they accept that the path to an agent modelling language is to be found through OO techniques. In that respect, Java is a sensible choice because, in addition, it provides a range of extra qualities.

Jini While Java has a whole host of features that make it suitable for the construction of agent-based systems, perhaps the most important development, that makes Java an almost ideal platform is the recent release of the Jini connectivity technology [1]. Essentially, Jini is intended to resolve the problem of network administration by providing an interface where different components of the network can join or leave the network at any time.

Such a collection of entities is called a Jini federation, and entities within the federation represent service providers or service consumers. The strength of the Jini system lies in the fact that it abstracts above the level of network protocols or device drivers by introducing this notion of a service. A service can be anything: a storage device, a printer, a camera or, more importantly in our case, a software component. When a service joins a Jini federation it brings along with it a set of attributes that describe its capa-

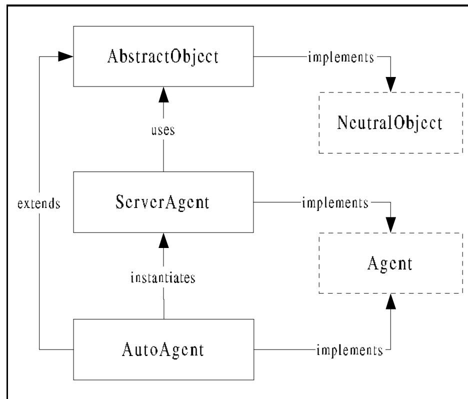

Figure 1. Paradigma Agent Hierarchy

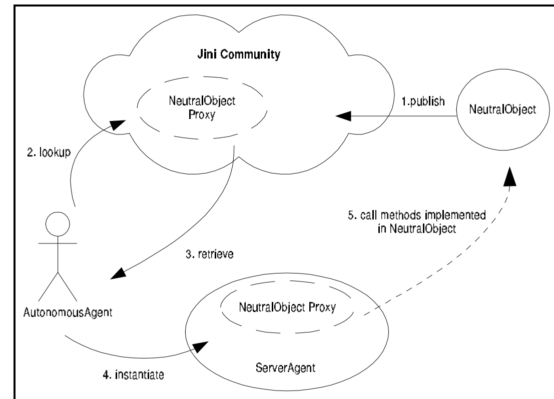

Figure 2. Autonomous Agent Functioning

bilities. These attributes can then be examined by a human or software agent to determine the service's suitability for achieving the desired goals of a potential service user. If the service is suitable then the agent of the potential user can negotiate an agreement under which the service provision will take place.

Furthermore, the system provides features that make it dynamic. Through the use of the notion of a lease, Jini services can indicate the period for which they would like to join a federation. At the end of the lease a service may choose to renew the lease or abandon the federation. (Scenarios in which a charge in proportion to the lease period is required to join are not difficult to envisage.) Another appealing aspect of Jini technology that is relevant to agent-based systems is that it is not necessary for a service that wishes to join a federation to have direct access to a JVM. Such a service can use a proxy, which mediates between a JVM and the actual component being controlled. This enables developers to easily implement Jini wrappers for legacy software or for devices that cannot support a JVM.

It should be immediately obvious how these ideas are connected to many of those arising from agent-based systems. In essence, Jini provides the plumbing. It relieves the developer from having to re-invent the wheel every time. As is identified by Bradshaw et al [2], such systems have been badly needed to push agent development forward.

4 Object and Agent Creation in Paradigma

Having introduced the main design decisions in Paradigma we now move on to show how the pieces fit together by illustrating the cycle of creation of a neutral object and an autonomous agent and the subsequent manipulation of the neutral object by an autonomous agent.

The most basic structure in Paradigma is the entity relationship that translates the four-tiered hierarchical framework. It separates the permanent and temporary entities in the world, as shown in Figure 1. While AutoAgent extends AbstractObject, which defines the behaviour of NeutralObject, ServerAgents have the functionality of Agents but instead of extending AbstractObject they wrap around it. Once the ServerAgent is no longer required the wrapped NeutralObject is released.

Neutral objects are entities in the framework which are described by their attributes and capabilities. In Paradigma, attributes are defined through an XML file with a distinction being made between static and dynamic attributes (which may change over time). Capabilities are similarly described in the XML file but are actually Java classes that are dynamically loaded into the neutral object's JVM at run-time. In this way, agent designers can have access to libraries of capabilities that they can automatically attach to their agents for particular agent functionality.

When a desirable neutral object is located by an agent it can be engaged directly by that agent to instantiate a server agent from it. Server agents are temporary agents that adopt goals from (or have goals ascribed to them by) the engaging agent, as illustrated in Figure 2. If a neutral object is already engaged there are several courses of action that might be followed. The requesting agent may simply transfer its own goal to a queue of goals the neutral object must execute and wait its turn. Alternatively, if it has a higher priority than the agent currently engaging the object, it may demand immediate attention. Finally, the requesting agent could negotiate with the target for an alternative agreement that meets both requirements, as described elsewhere [8].

We have discussed the nature of server agents, but have not detailed the exact mechanism through which users can interact with the neutral object, and we examine the three principal ones below.

Execution in the engaging agent's JVM In this case the entire neutral object implementation is published to the Jini community. Thus, when it is downloaded by the engaging agent, all links with neutral object's JVM are lost, and

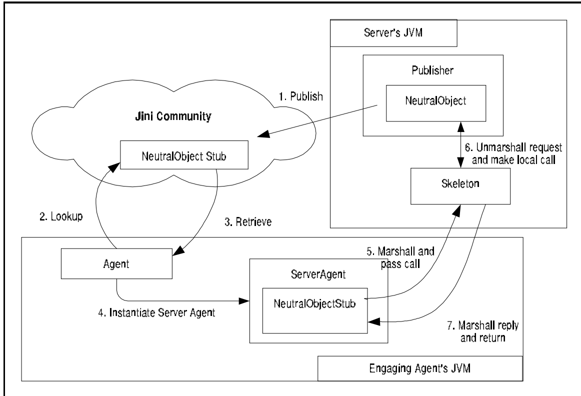

Figure 3. Execution in engaging agent's JVM

calls to the newly engaged server agent (the orginal neutral object) are executed locally. This behaviour is useful if the available service represents a required extension to the agent engaging the neutral object, e.g. a network administration agent that requires a new module to control a device.

Execution takes place on the server's JVM As shown by Figure 3, a more sophisticated approach is for the neutral object to publish a proxy (an RMI stub[1]) to its implementation, which remains in the neutral object's JVM. As far as the engaging agent is concerned, the behaviour is the same as the previous case in that it also instantiates a server agent to wrap the neutral object (stub). All calls are, however, communicated to the original engaged neutral object's remote JVM. This is useful in instances where the neutral object serves as a channel to information on a database or as an interface for a remote sensor, or for the purposes of distributing computing power.

Execution takes place on both client and server Here we have what is often called a *smart proxy*. The neutral object publishes an implementation of its interface where some of the processing is done in the engaging agent's JVM and some on the engaged agent's (the oiginal neutral object's) remote JVM. Such an implementation is useful in instances where it is more profitable to distribute the processing over both machines, e.g. in the case where the engaging agent can better perform local processing (due to a more powerful processor) of the information obtained by a remote call to the neutral object.

In the current implementation of Paradigma, execution takes places remotely as described in the second case above, simply because we aim to focus on the distribution of computing and control of remote devices.

Now, agents act according to their goals, and autonomous agents can generate goals internally or they can adopt goals from others. In contrast, neutral objects have goals ascribed to them and become server agents. In any case, agents attempt to form plans to satisfy currently active goals. At present, planning is not very flexible as plans are not generated on-line but taken from a repository of plans to get the most suitable plan for achieving the current goal, as in many BDI architectures (eg. [9]).

Autonomous agents give meaning to agent-based systems through their ability to generate their own goals, driven by their motivations. Motivations along with an agents goals and plans thus completely define the agent's behaviour. On instantiation of an autonomous agent, XML files are required for attributes, capabilities, motivations, goals and plans. The sets of goals and plans form, respectively, the *goal base* and *plan base* of the agent. Based on motivations, an autonomous agent chooses the goal that offers the greatest motivational utility, leading to an attempt to select and execute a plan that involves certain agents capabilities or the engagement of other agents. Once a goal has been achieved, or if attempting to achieve it any longer is futile, autonomous agents modify their motivations accordingly and choose another goal from the goal base.

5 Agent Interaction

Until now, we have discussed the creation and instantiation of the various components of the agent framework, but have not considered how autonomous agents can interact with each other. In Paradigma, an autonomous agent that chooses to make itself available for cooperation with other autonomous agents needs to join a Jini community so that it can be *discovered*. Once it discovers another, they enter into negotiation to agree on some sort of service provision, and this negotiation requires a base communication infrastructure. There are several possibities for enabling the communication required for interaction between agents, ranging from object-based communication with RMI, to socket-based XML messages. For more robust behaviour a third-party message router could be used. It is important, however, that the framework provides a layer of abstraction above the underlying communication method so as not to tie the framework to any particular way of communication.

Figure 4 illustrates the issues discussed in this section as well as showing how they come together with the overall functionality of Paradigma. An autonomous agent, Jamal, wishes to access a database in order to gain some information. Jamal engages a neutral object that acts as an interface to the database, and transfers its goals to it, thus instantiating a server agent. In the meantime, Jamal makes itself available for contact by other agents through the publication

[1] An RMI stub takes care of *marshalling* the parameters that are passed to the neutral object and sends them across the network. At the other end, an RMI skeleton will *unmarshall* the parameters and make the call locally. The results are then marshalled, again through the skeleton, and passed to the stub that will unmarshall them and return the results to the client that made the call. The most important implication is that all information should be serializable so that it can be transferred over the network.

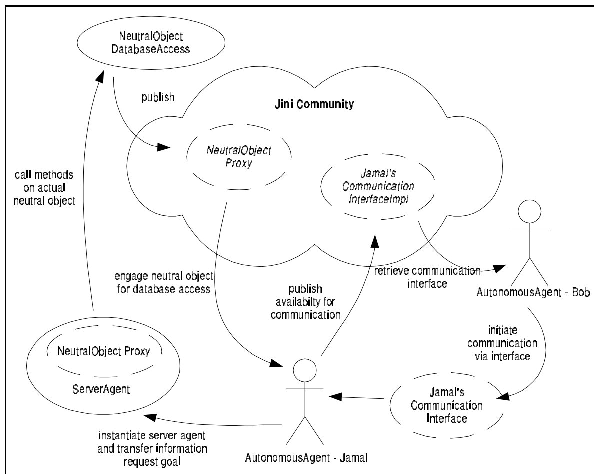

Figure 4. Agent system snapshot

of a communication interface implementation. Another autonomous agent, Bob, wishes to communicate with Jamal, and does so by downloading the implementation of the communications interface.

6 Conclusions

Several issues have not been addressed in this paper, partly due to space constraints, partly due to focus, and partly because they have not yet been tackled in the context of Paradigma. We should note, however, that the scope for extending this work to address many other such issues is considerable. For example, the issue of payment for services rendered by agents, whatever the form of the engagement, is likely to be a key concern for the broader acceptance of systems developed in this way. Jini provides for such cases via its leasing mechanisms, which we have only briefly touched upon in discussing the use of leasing for joining a Jini community, and its role in making a system more dynamic by automatically cleaning up services not renewing their leases. However, services themselves can grant leases through the same mechanisms to consumers of their resources including, for example, the payment of some amount towards the use of the service.

As discussed elsewhere [5], we are entering a new phase of agent research and development in which the focus must be less on the exciting and inspirational issues, and more on the mundane but fundamental issues of consolidation that underlie any serious technological effort. These include the integration with, and use of, existing technology that is tried and tested, the application of agent solutions to pre-existent problems, the linkage of agent theory and practice, and the augmentation of the technology with facilities for development. These are exactly the issues that we are beginning to address through the Paradigma implementation environment described in this paper.

Paradigma has been developed as a result of very practical concerns — supporting the development of agent-based systems. It has been constructed using appropriate underlying technologies that are becoming increasingly widespread and largely standardised. In this sense, the applicability and accessibility of the work is unquestionable, since it engages directly with the broader development community. Yet, at the same time, the work is underpinned by a concern for principled development, and Paradigma in fact implements a sophisticated and well-developed foundational formal agent framework within which strong theoretical analyses are facilitated. Paradigma thus satisfies the needs of several different audiences and, though still early in its development, suggests a likely avenue to explore for effective and valuable progress.

References

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

[2] J. Bradshaw. Agents for the masses. *IEEE Intelligent Systems*, 14(2):53–63, 1999.

[3] M. d'Inverno and M. Luck. Development and application of a formal agent framework. In M. G. Hinchey and L. Shaoying, editors, *ICFEM'97: First IEEE International Conference on Formal Engineering Methods*, pages 222–231. IEEE Press, 1997.

[4] D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, LNAI 1038*, pages 56–71. Springer-Verlag, 1996.

[5] M. Luck. From definition to development: What next for agent-based systems. *Knowledge Engineering Review*, 14(2):119–124, 1999.

[6] M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 254–260. AAAI Press / MIT Press, 1995.

[7] M. Luck and M. d'Inverno. Engagement and cooperation in motivated agent modelling. In *Proceedings of the First Australian DAI Workshop, Lecture Notes in Artificial Intelligence, 1087*, pages 70–84. Springer Verlag, 1996.

[8] M. Luck and M. d'Inverno. Motivated behaviour for goal adoption. In *Multi-Agent Systems: Theories, Languages and Applications — Proceedings of the Fourth Australian DAI Workshop, Lecture Notes in Artificial Intelligence, 1544*, pages 58–73. Springer Verlag, 1998.

[9] A. S. Rao. Agentspeak(l): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, LNAI 1038*, pages 42–55. Springer-Verlag, 1996.