Agent Factory: Generative Migration of Mobile Agents in Heterogeneous Environments

F.M.T. Brazier, B.J. Overeinder, M. van Steen, and N.J.E. Wijngaards
Department of Computer Science, Vrije Universiteit Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{frances,bjo,steen,niek}@cs.vu.nl

ABSTRACT

In most of today's agent systems migration of agents requires homogeneity in the programming language and/or agent platform in which an agent has been designed. In this paper an approach is presented with which heterogeneity is possible: agents can migrate between non-identical platforms, and need not be written in the same language. Instead of migrating the "code" (including data and state) of an agent, a blueprint of an agent's functionality and its state is transferred. An agent factory generates new code on the basis of this blueprint. This approach of *generative mobility* not only has implications for interoperability but also for security, as discussed in this paper.

Keywords

mobile agents, process migration, compositional design

1. INTRODUCTION

In a global, distributed computer infrastructure, in which the Internet provides connectivity, mobile agents are seen as a promising computational approach to distributed computing, resource management, and security.

Mobile agents allow for computations to dynamically adapt to a changing environment, for example, by migrating from one machine to another. The decision to migrate is most oftern taken autonomously by the mobile agent itself. The ability of migration provides mobile agents a means to overcome the high latency or limited bandwidth problem of traditional client-server interactions by moving the computation to required resources or services. The current evolution of intelligent and active networks in system and network management, for example, is based on this technology. A similar tendency is observed in the search and filtering of globally available information such as in the electronic marketplaces, e-commerce, and information retrieval on the World Wide Web [8].

To support agent mobility a distributed system needs provisions to physically migrate units of computation at runtime. This migration includes relocation of an agent's code base and state to another platform. Code and state migration is a complex task with technical complications such as binary incompatibility of two heterogeneous platforms. The current solution to migration of active units of computation is to provide homogeneous platforms, either physically such that binary checkpoints can be restarted at another location, or virtually by using virtual machines, e.g., the Java Virtual Machine, providing a machine independent platform. The homogeneity requirement, physical or virtual, is a strong requirement: mobility is otherwise impossible.

Another important issue in mobile agents technology is security. In most current systems trust in the owners and in the machines on which an agent has previously run, are the basis for a security model. Code signing and certificates are the techniques used to this purpose.

This paper presents a completely new approach to agent mobility. Not the code migrates, but an agent's blueprint and state. A receiving platform *regenerates* a mobile agent as it migrates to its new location. Homogeneity is no longer required: an agent programmed in Java can be transformed to, for example, a Python implementation of the agent with the same functionality. Trust in an agent coming from another machine increases considerably if the receiving platform uses its own trusted components to reconfigure an agent.

Section 2 discusses mobility of processes and agents in more detail. Section 3 describes the concept of an agent factory. Heterogeneous migration based on this concept is the topic of Section 4. Implications of this approach for heterogeneous migration and security are discussed at more length Section 5. Section 6 sums up the results and proposes future research.

2. BACKGROUND

An agent in a mobile agent system is typically associated with a unit of computation which resides in the lower layers of a virtual machine. A unit of computation is composed of the code describing its behaviour, the data associated with it, and its execution state. Mobile agent systems allow migration of the whole unit or a part thereof, i.e., one or more of the three constituents mentioned above. The most relevant differences among existing systems lie exactly in what is moved and how [18].

A distinction can be drawn based on whether the execution state is migrated along with the unit of computation or not. Systems providing the former option are said to support *strong mobility*, as opposed to systems that discard the execution state across migration, and are hence said to provide *weak mobility*. In systems supporting strong mobility, migration is completely transparent to the migrated program, whereas with weak mobility, extra programming is required in order to manually save part of the execution state.

Strong mobility as found in NOMADS [22], Ara [16], and D'Agents [7], requires that the entire state of the agent, includ-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC 2002, Madrid, Spain
Copyright 2002 ACM 1-58113-445-2/02/03...$5.00.

ing its execution stack and program counter, is saved before the agent is migrated to its new location. This process of saving the entire state of an executing process is called *checkpointing*. An important quality of strong mobility is transparent migration of the running process. That is, the agent is not aware of the migration and bindings to other agents and objects are transparently resolved, i.e., references to agents and objects are location independent. The checkpoint/migration facility can be either implemented at the operating system level [10, 15, 9] or can be incorporated within the virtual machine of an interpreted language (e.g., within the Java Virtual Machine [22]).

Despite the advantages of strong mobility, many agent systems support weak mobility (like Ajanta [23] and Aglets [11]). Most of the agent systems are implemented on top of the Java Virtual Machine (JVM), which provides with object serialization basic mechanisms to implement weak mobility. The JVM does not provide mechanisms to deal with the execution state.

Agent mobility is, in general, most easily realized in homogeneous environments. For strong mobility with checkpoint/migration incorporated at the operating system level, agent mobility is limited to identical computer architectures running the same operating system. Agent mobility facilities implemented at the virtual machine level makes the migration of agents machine independent, but is still homogeneous in language, i.e., only migration of agents from Java to Java platforms.

Migration of mobile agents does not need to be constrained by homogeneity of code bases and platforms. Agents can be migrated across heterogeneous code bases and platforms by reconfiguration of the agents upon arrival at a new location. Blueprints of the functionality of an agent are the basis of the migration. At a new location, the agent is regenerated according to this blueprint using components specific to the local agent platform. The functional components can be from another code base than the originating agent, but also the agent platform can differ; hence interoperability between agent platforms can be realized. The next section describes the means with which this can be achieved: an agent factory.

3. AGENT FACTORY APPROACH

Assuming agents have a compositional structure described by their *blueprints*, building an agent is, in fact, a configuration task: a task that can be automated. Automated (re-)design of agents is the task of an agent factory [3]. This section describes an existing agent factory, one of the services of the AgentScape framework.

Section 3.1 defines the concept of blueprints in more detail. Section 3.2 discusses characteristics of an agent factory, and Section 3.3 describes a current prototype of this agent factory.

3.1 Blueprints

In the following discussion, it is assumed that agents are designed to have a compositional structure. A blueprint is a high level specification of the functionality and operational semantics of an agent. The specification describes the behaviour of an agent in terms of its basic building blocks: components, control flow and data flow. The resulting blueprint is expressed in a high level specification language: the blueprint language. (The blueprint specification language is somewhat similar to, for example, Very High Design Language (VHDL) used in VLSI design.)

Agent factories interpret the agent blueprint and generate executable code from, for example, Java, Python, or C components. By interpreting the blueprint language and generating executable code, agent factories conceptually provide a high level virtual machine for the blueprint agents. The operation of agent factories can to some extent be compared with Java to native machine code compilers. For example, at arrival at a host, a mobile Java agent can be compiled to native machine code using the Java class implementations from the local repository. An agent factory is more flexible than a "standard" interpreter, as the agent factory is able to generate new blueprints for new agents, using knowledge bases.

The concept of a building block is used to describe the components within an agent's blueprint at two levels of abstraction: conceptual and detailed/operational. At each level of abstraction the behaviour of the agent is described. Some building blocks contain open slots, others are fully specified and operational. Both define their functionality on the basis of their interfaces. Open slots define the interfaces of the building blocks to be inserted.

Depending on availability and domain of application libraries of building blocks may include: partial agent designs (cf. generic models/design patterns [5, 17, 19]), knowledge-based models (e.g., problem-solving models [21] or generic task models [2]), agent-wrappers (providing cross platform interfaces) (e.g., AgentScape, Zeus [13], message parsing Ajanta [23]), et cetera. Building blocks may be written in, e.g., UML, Python, C++, CommonKads, etc.

3.2 Characteristics of an agent factory

Whether the need for adaptation is identified by an agent itself, or by another agent, is irrelevant in the context of this paper. An agent factory simply constructs new agents and/or modifies existing agents [3]. The (re-)design of agents is fully automated, with very limited interaction with outside parties. The concept of an agent factory requires (i) agents to have a compositional structure, (ii) one or more libraries of re-usable agent components, and (iii) one or more ways to describe the functionality of these agent components.

In the agent factory discussed in this section three additional assumptions hold: (i) two levels of description if an agent's behaviour are distinguished: conceptual and detailed, (ii) no commitments are made to specific programming languages and/or ontologies, and (iii) a shared blueprint language can be defined.

3.3 Agent factory prototype

A first prototype of the agent factory automatically (re-)designs an information retrieval agent: its blueprint and executable code. The information retrieval agent is based on an agent architecture, shown in Fig. 1.

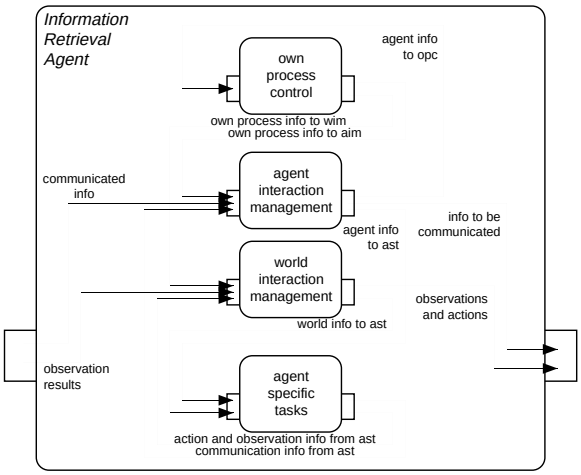

Figure 1: Architecture of a simple information retrieval agent.

In this prototype, conceptual components are specified in the DESIRE framework [1, 2]. The compositional nature of DESIRE models, and the separation between processes and knowledge makes

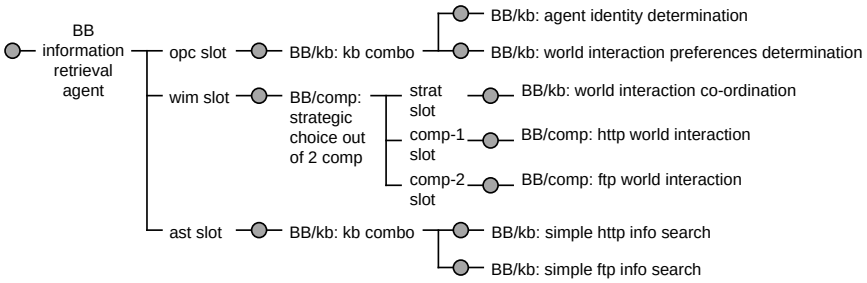

Figure 2: Building block configuration of a simple information retrieval agent.

it possible to specify knowledge intensive systems from reusable components. A structure-preserving mapping exists between the configuration of building blocks at the conceptual level of abstraction and the configuration of building blocks at the detailed level of abstraction. The detailed components are in Java.[1]

This prototype agent factory itself is written in Java, and contains enough knowledge to be able to (re-)design simple information retrieval agents. Figure 2 illustrates a building block-configuration in which two levels of building blocks were required: each open slot required a building block that itself contained other open slots. Note that the lower level building blocks make a distinction between open slots for data, and open slots for processes.

4. MIGRATION USING THE AGENT FACTORY SERVICE

One of the strengths of the agent factory concept is that it provides a means to support migration of agents in heterogeneous environments that require a high level of security. Section 4.1 discusses pre-conditions for successful migration of agents. Section 4.2 describes the approach in agent-factory-enhanced migration.

4.1 Migration pre-conditions

To facilitate the description of migration of an agent, it is assumed that an agent consists of executable code and state. Executable code may contain "code and data," if these can be distinguished, or may be inseparable (as with Prolog). When an agent migrates, it needs to retain sufficient information from its state to resume execution at its destination. Note that this description leans towards weak-mobility: it may not be necessary to transport the entire state of an agent.

Although it is not necessary for the source and destination host to both have access to an agent factory, it greatly simplifies descriptions of the migration process. An agent needs to be able to store and restore information on its state; this is a requirement for interoperability. Possibly an implementation-independent format such as XML, RDF or OIL may be used.

The agent factories on the hosts need to share some building blocks. E.g., each agent factory may have the same libraries of building blocks at the conceptual level of abstraction, but may have different libraries of building blocks at the detailed level. For example, an agent factory may have a mapping from a conceptual agent architecture building block to a detailed building block written in Java; while another agent factory may have a detailed building block written in C++.

[1]Automated prototype generation within the DESIRE framework on the basis of detailed formal specifications facilitates verification and validation of knowledge intensive systems; this feature is not used within the current prototype of the agent factory.

4.2 Approach to migration

In essence, migration entails moving an agent from one machine to another. This usually involves pre-packaging an agent before its move, such that it its executable code and state may be restored at the destination host. Migration using an agent factory diverges from standard mobility of agents in that executable code with state is *not* migrated, but the agent's blueprint together with (parts of) the agent's state. This might seem to be similar to Java agents and their interaction with class loader objects. A class loader object allows specific implementations of Java classes to be loaded. However, the approach described here is to migrate a specification of an agent that can be targeted to an implementation language like Java, Python, or Prolog.

Consider the following scenario for heterogeneous mobility, depicted in Fig. 3. An information retrieval agent A currently resides on a host machine H1. This host runs the Ajanta [23] agent platform, and, as such, supports Java agents. The agent wishes to move to another host: host H2. The host H2 runs the DESIRE platform, and its agents run code generated by the DESIRE platform.

In the process of migrating the agent A from host H1 to host H2, the agent first needs to offload information on its state. Then the agent factory on host H1 sends the blueprint of the agent, together with the state information of the agent to host H2.

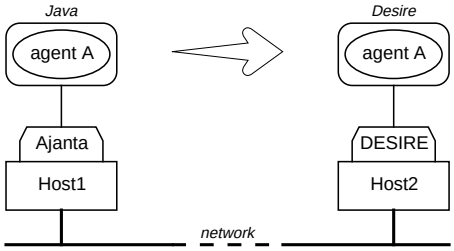

Figure 3: Example migration scenario in which agent A on host 1 (written in Java, running on Ajanta) migrates to host 2 (where it will be specified in DESIRE and running on DESIRE).

Host H2's local agent factory receives the blueprint of the agent and state information. This agent factory designs a DESIRE agent A on the basis of the blueprint of agent A. This DESIRE agent A (i.e., a functionally equivalent incarnation of the Java agent A) runs on DESIRE's virtual machine (the DESIRE-interpreter), and is able to incorporate information on its state.

The agent factory on the receiving side regenerates the agent, possible in a different implementation language and in a different environment. The agent may need to acquire information about its new environment and react to changes.

5. ISSUES

Heterogeneous migration of agents, possibly across agent platforms, raises a number of issues with respect to interoperability (Section 5.1) and security (Section 5.2).

5.1 Heterogeneity & interoperability

Migration using an agent factory makes it possible to migrate agents not only in a homogeneous environment, but also in heterogeneous environments. The executable code of an agent usually contains a part that provides the interfaces between the agent and the agent platform on which the agent "lives." Taking this interface into account, the following migration scenarios can be distinguished.

Homogeneous migration An agent migrates to another host without any changes to the format of its executable code or the interfaces to the agent platform. This form of migration requires that source and destination platform offer the same interfaces, but also that the (virtual) machine that executes the agent is the same at both sides. In practice, this form of migration is most common.

Cross-platform migration An agent is migrated to another host with a different agent platform, but that offers the same (virtual) machine architecture. This generally entails changes to the interface to the agent platform, but not necessarily changes to the format of its executable code. This form of migration may occur when, e.g., a Java-agent migrates from a Ajanta platform to a Zeus platform. One commonly applied solution is to offer wrapper interfaces that hide the differences between source and target platform. Another approach, followed in MAF [14] or FIPA, is to enforce platforms to implement a standard interface for interoperability.

Agent-regeneration migration An agent migrates to a host running a different (virtual) machine requiring that the agent is regenerated, resulting in different executable code. Note that the target agent platform may be the same as that of the source, which may simplify regeneration. To regenerate an agent, it is necessary that the target has a blueprint of the agent. We are not aware of agent systems that support this approach.

Heterogeneous migration An agent migrates to another host with a different agent platform and offering a different (virtual) machine. In this case, regeneration of the agent is necessary. Because the underlying agent platform is also different the agent's blueprint must be platform independent, which may complicate matters.

This paper advocates heterogeneous migration as it offers most flexibility. As distributed systems are gradually required to scale worldwide across different administrative organizations, and to support a myriad of platforms, solutions are needed that anticipate heterogeneity and adaptability. Regeneration of agents for different underlying platforms is a step towards meeting such requirements.

The approach described in this paper combines heterogeneous migration with weak migration. The term proposed for our approach is *generative migration*. Generative migration for agents may open the world of distributed systems to agent-developers. The adage "write once, run everywhere" is achieved while retaining heterogeneity and tackling the problem of interoperability.

Generative migration requires that a target host has access to an agent factory capable of generating an agent for that target. Ideally, this factory is placed on the target host, or otherwise available on the same local-area network. An important issue is that the factory is trusted to generate an agent that the target can trust. Security and trust are briefly discussed below.

Our approach has the additional benefit that various optimizations become possible. For example, the agent generated by a factory may be optimized with respect to the target's machine architecture, or the way that local resources such as databases are accessed. In addition, it is to be expected that transmission of an agent's blueprint and information on its current state will generally require less network resources than migrating an agent using more traditional approaches. On the downside, the agent-generation process may affect overall performance in the case of often-migrating agents.

5.2 Security

Migration of an agent involves security from a number of perspectives. Security issues related to authenticating an agent, and deciding whether an agent is allowed to migrate to its destination, are not discussed in this paper. What remains are how to protect an agent against attacks during and after its migration, and how to protect a target against attacks from a malicious agent. Considerable research has already been conducted with respect to both issues and which can be applied to our approach. In the following, the role of security is briefly considered. It should be noted, however, that security in our approach is subject to further research.

5.2.1 Protecting an agent

A mobile agent may be preyed upon in transit, or while running on a malicious host. It is impossible to protect an agent against modifications during its transfer or execution in an untrusted environment [4]. At best, it can check whether an agent has been maliciously modified and take appropriate measures after the fact. Our approach to migration can help here.

It is important to realize that an agent's blueprint does not change during its lifetime. (Except for reconfiguration at an agent factory.) Consequently, by adding an integrity check to a blueprint using standard techniques for digital signatures [20], it is easy to detect whether a blueprint has been changed. When a factory notices that a blueprint has been changed, it can either discard the agent or generate it from the original blueprint. The latter is possible only if that blueprint is locally available, or if it can be retrieved in a secure way. Securely retrieving a blueprint requires that a factory can set up a secure channel to a blueprint repository, that is, a channel that provides authentication and transmission integrity.

Of course, it should be possible to support *evolutionary agents* for which new blueprints are generated. However, blueprint generation should be done only by trusted factories and never as a solution to migration. As such, it falls outside the scope of this paper.

5.2.2 Protecting a host

A host that admits foreign mobile agents to its resources takes a risk: some of the agents may be malicious, and may try to subvert (parts of) the host. The problem with traditional approaches to agent migration is that it is impossible to check in advance whether or not imported code does only what it promises. The solution is to construct what are known as sandboxes [24]: a restricted environment in which, effectively, each instruction is monitored and checked before being executed. If access to resources is violated, execution halts. The sandbox model is quite restrictive, and has been extended since its initial introduction (see, for example, [6, 12]).

Regenerating agents from blueprints may considerably help in

protecting a host against malicious code. Normally, blueprints do not contain code descriptions, but refer only to interfaces and components that should be locally available to an agent factory. The code contained in these components may have been verified by the owner of the factory, or have been obtained from trusted sources. Of course, protection will fail if verification has not been done properly. In effect, a requirement is that trusted code is available before an agent migrates to a target, or that can be retrieved from a trusted repository through a secure channel.

In those cases that blueprints require execution of untrusted code, traditional approaches based on sandboxing techniques or protection domains need to be implemented as part of the target platform.

A mobile agent arriving at the host is regenerated on the basis of its blueprint, using only detailed building blocks which the host approves of. Although the specific configuration of building blocks may be new to the host, a number of security risks can be removed. The mobile agent may still be untrustworthy, but is prevented from executing certain calls on the host.

As an example, consider a bank that wishes to use mobile agents which may transact money from one account to another. The bank offers libraries of building blocks written in Java to its clients. These clients may build mobile agents that can perform transactions at the bank. The bank admits only those mobile agents that can be regenerated on the basis of their blueprint using building blocks written in Cobol. This may give more confidence to the bank that the mobile agents will not be able to tamper with their system. Note that cheating, using other people's passwords and certificates is not necessarily stopped by this approach.

6. DISCUSSION AND FUTURE WORK

Agents, and in particular mobile agents, offer a means for application developers to build distributed applications. Mobility of agents is often required for various reasons, notably performance. Current agent platforms offer a wide range of services to agent developers, including mobility. However, mobility of agents is usually limited to hosts running the same agent platform and that have the same (virtual) machine architecture. In other words, it is often restrained to a homogeneous environment.

The approach described in this paper transcends this homogeneity and proposes *generative mobility*. In generative mobility, a blueprint of an agent's functionality is transported, together with information on the agent's state. At its destination, an agent factory regenerates the executable code of the agent on the basis of its blueprint. An agent may then restore its state and resume execution.

With generative mobility an agent may travel to locations that offer a different platform and that require it to adopt a different (virtual) machine architecture. In other words, generative mobility supports true heterogeneous mobility, offering an agent maximum flexibility with respect to where it wants to go. In addition, an agent's executable code can be optimized for its destination, while retaining its required agent-level functionality. In their own way, agent factories and blueprints offer a language and agent platform independent virtual machine that allows for heterogeneous migration.

Agent factories play an important role in generative mobility as they offer the services needed to generate executable code on the basis of blueprints. Agent factories rely on libraries of building blocks from which agents can be configured. As a consequence, agent factories need to share these (conceptual) building blocks to understand an agent's blueprint and be able to generate its associated executable code. Homogeneity in agent architectures is a likely consequence of this approach.

Research on generative mobility is clearly not finished. In particular, the use of blueprints needs to be investigated to determine to what extent blueprints are flexible enough to describe agents, and how security can be adequately dealt with. Agent factories form an important component within our worldwide distributed AgentScape system that allows agents to be automatically (re-)designed. Currently a prototype of the agent factory (namely the libraries of components) is being built that supports generative mobility.

The use of generative mobility for relatively closed environments, such as hospitals, is currently being studied. Generative mobility with trusted code libraries on the hospital side may possibly provide a solution to controlled access to medical dossiers. Insurance companies, for example, are allowed limited access to specific types of information and processing. Control over the executable code of an insurance company's agent provides a means for a hospital to control the calls and data an agent may execute inside the hospital.

Acknowledgments

This research is supported by NLnet Foundation, http://www.nlnet.nl. The authors wish to acknowledge the contributions made by Hidde Boonstra, David Mobach, Oscar Scholten and Sander van Splunter.

7. REFERENCES

[1] F.M.T. Brazier, B.M. Dunin-Keplicz, N.R. Jennings, and J. Treur. Formal specification of multi-agent systems: A real-world case. *International Journal of Co-operative Information Systems*, 6:67–94, 1997. Special Issue on Formal Methods in Co-operative Information Systems: Multi-Agent Systems.

[2] F.M.T. Brazier, C.M. Jonker, and J. Treur. Principles of component-based design of intelligent agents. *Data and Knowledge Engineering*, 2002. In press.

[3] F.M.T. Brazier and N.J.E. Wijngaards. Automated servicing of agents. In *Proceedings of the AISB-01 Symposium on Adaptive Agents and Multi-Agent Systems*, pages 54–64, March 2001.

[4] W.M. Farmer, J.D. Guttman, and V. Swarup. Security for mobile agents: Issues and requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, Baltimore, MD, October 1996.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994.

[6] L. Gong and R. Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 125–134, San Diego, CA, March 1998.

[7] R.S. Gray, G. Cybenko, D. Kotz, R.A. Peterson, and D. Rus. D'Agents: Applications and performance of a mobile-agent system. *Software: Practice and Experience*, 2001. In press.

[8] V.N. Gudivada, V.V. Raghavan, W.I. Grosky, and R. Kasanagottu. Information retrieval on the World Wide Web. *IEEE Internet Computing*, 1(5):58–68, September/October 1997.

[9] K.A. Iskra, F. van der Linden, Z.W. Hendrikse, B.J. Overeinder, G.D. van Albada, and P.M.A. Sloot. The implementation of Dynamite: An environment for migrating PVM tasks. *Operating Systems Review*, 34(3):40–55, July 2000.

[10] D. Johansen, R. van Renesse, and F.B. Schneider. Operating system support for mobile agents. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 42–45, Orcas Island, WA, May 1995.

[11] D.B. Lange, M. Oshima, G. Karjoth, and K. Kosaka. Aglets: Programming mobile agents in Java. In *Worldwide Computing and Its Applications*, volume 1274 of *Lecture Notes in Computer Science*, pages 253–266. Springer-Verlag, Berlin, Germany, 1997.

[12] D. Malkhi and M. Reiter. Secure execution of Java applets using a remote playground. *IEEE Transactions on Software Engineering*, 26(12):1197–1209, December 2000.

[13] H. Nwana, D. Ndumu, L. Lee, and J. Collis. ZEUS: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal*, 13(1):129–186, 1999.

[14] OMG. Mobile agent facility specification. OMG Document formal/00-01-02, Object Management Group, Framingham, MA, January 2000.

[15] B.J. Overeinder, P.M.A. Sloot, R.N. Heederik, and L.O. Hertzberger. A dynamic load balancing system for parallel cluster computing. *Future Generation Computer Systems*, 12(1):101–115, May 1996.

[16] H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (MA'97)*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, April 1997. Springer-Verlag.

[17] F. Peña-Mora and S. Vadhavkar. Design rationale and design patterns in reusable software design. In *Artificial Intelligence in Design (AID'96)*, pages 251–268, Dordrecht, 1996. Kluwer Academic Publishers.

[18] G. P. Picco. Mobile agents: An introduction. *Microprocessors and Microsystems*, 25(2):65–74, April 2001.

[19] A.J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, Reading, MA, 1996.

[20] B. Schneier. *Applied Cryptography*. John Wiley, New York, NY, 2nd edition, 1996.

[21] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. Van de Velde, and B. Wielinga. *Knowledge Engineering and Management, the CommonKADS Methodology*. MIT Press, 1999.

[22] N. Suri, J. Bradshaw, M.R. Breedy, P.T. Groth, G.A. Hill, and R. Jeffers. Strong mobility and fine-grained resource control in NOMADS. In *Proceedings of the Joint Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA 2000)*, pages 2–15, Zurich, Switzerland, September 2000.

[23] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh. Mobile agent programming in Ajanta. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 190–197, Austin, TX, May 1999.

[24] D.S. Wallach, D. Balfanz, D. Dean, and E.W. Felten. Extensible security architectures for Java. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 116–128, St. Malo, France, October 1997. ACM.