View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Unisa Institutional Repository

Architectural Components for the Efficient Design of Mobile Agent Systems

MARTHIE SCHOEMAN
AND
ELSABÉ CLOETE
University of South Africa

Over the past eighteen months, there has been a renewed interest in mobile agent technology due to the continued exponential growth of Internet applications, the establishment of open standards for these applications, as well as the semantic web developments. However, the lack of a standardised programming model addressing all aspects of mobile agent systems prevents widespread deployment of the potentially useful technology. The architectural requirements dealing with all aspects of a mobile agent system are not clearly stipulated. As a result, the commercially available mobile agent systems and mobile agent tool kits address different mobile agent issues, and little reuse of available technologies and architectures takes place. The purpose of this paper is to describe an architectural model that identifies the components representing the essential aspects of a mobile agent system. Due to the intensive nature of development, implementation and testing of this model, we describe preliminary work. However, in the meanwhile, there are benefits associated with this preliminary model, namely that it provides a clear understanding of the architectural issues of mobile agent computing, giving novice researchers and practitioners who enters the field for the first time a foundation for making sensible decisions when researching, designing and developing mobile agents. The model is also significant in that it provides a benchmark for researchers and developers to measure the capabilities of mobile agents created by commercially available tool kits.

Categories and Subject Descriptors: H.1[**Information Systems**]: Models and Principles - *General*; D.0 [**Software**]: General
General Terms: Design; Standardisation
Additional Key Words and Phrases: Mobile Agent Systems, Software architecture model

1. INTRODUCTION

In 1999, Kotz and Gray [1999] predicted that many major Internet sites will accept mobile agents within a few years, while Milojicic [1999] claimed that although mobile agent technology has raised considerable interest in the research community and in industry, the promised deployment has not yet and may not even emerge. The number of publications on the topic between 1996 and 1999 and the subsequent noticeable decline between 1999 and 2001 support Milojicic's point. However, since the end of 2001, there has been a considerable rise in the number of publications again. The renewed interest is largely due to the recent materialization of the semantic web [Berners-lee et al. 2001, Kagal et al. 2003], as well as the continued exponential growth of Internet applications and the establishment of open standards for these applications.

The specific (previous) problems associated with mobile agent technologies have not yet been resolved. According to Lange [1997], the lack of a programming model for agent-based applications prevents wider mobile agent deployment. Kotz et al. [2002] corroborated this by saying that few, if any, of the current mobile agent systems will be able to meet the needs of large, complex applications or be able to provide design paradigms for such applications. All-embracing programming standards for the mobile agent paradigm are almost non-existent. Kotz et al. [2002] warned that if mobile agents are to have significant impact on the Internet and mobile computing, the identification of the key features that make mobile agents efficient and contribute to the possibility of successful deployment are important. These features ought to be extracted to form a coherent, flexible set of standards.

While it may not be an overly complex task for a systems programmer to build and deploy a mobile agent or a mobile agent system, practitioners who do not work with agents regularly need guidance to develop them. Kendall et al.[2000] reason that agent development to date has been done independently, leading to various problems such as the lack of an agreed definition, duplication of effort, inability to satisfy industrial strength requirements and incompatibility. Furthermore, in a detailed literature review [Schoeman 2003], we found that little reuse of agent architectures, designs or components seem to be taking place. We briefly mention a few examples depicting the typical method of reuse that were found in other instances. In one example both SMART [Wong et al. 2001] and the Distributed Management Framework [Ferridun & Krause 2001] implemented and extended the Objectspace Voyager™ platform capabilities to satisfy their application needs. In another example, Wang et.al [2000] extended the Aglet technology and

Author Addresses:
M. A Schoeman, Department of Computer Science and Information Systems, University of South Africa, P O Box 392, UNISA, 0003, South Africa; schoema@unisa.ac.za.
E. Cloete, Department of Computer Science and Information Systems, University of South Africa, P O Box 392, UNISA, 0003, South Africa; cloete@unisa.ac.za.
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 the copies are not made or distributed for profit or commercial advantage, that the copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than SAICSIT or the ACM must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2003 SAICSIT

Proceedings of SAICSIT 2003, Pages 48 – 58

implemented a number of additional high-level design principles to make it easier to program general agent applications. Aridor & Lange [1998], Kendall et al. [2000] and Tahara et al.[1999] all describe ways in which patterns can be used to assist in mobile agent designs. However, in general, they do not use the same names to describe (often similar) patterns, thereby enhancing the impression that no reuse takes place. Besides a few similar cases, the overall impression is that developments are done independently and that there is a lack of reuse.

Building mobile agent systems is a challenge because few guidelines exist on the topic and also because exhaustive testing is virtually impossible. The research question we are addressing in this paper is as follows: What are the architectural requirements for designing and developing a mobile agent system quickly and efficiently in order to make it possible to include different mobile agent characteristics, with maximum reuse of available technologies and architectures? In addressing this question, our purpose is to describe the architectural components representing the essential aspects of a mobile agent system in a simplified, general but clear way. In our quest to establish such an architectural basis, we have studied various mobile agent implementations, and extracted features and characteristics that are embedded in these agent systems. In a further comparative study that we conducted, we were also able to extract design and development guidelines and integrate them into an architectural model that represents the essential features of a mobile agent system.

The architectural model presented in this paper is of a theoretical nature, with its components deduced through extractions from practical implementations and a small number of tests that we performed on certain features. The purpose is not to report on the tests of individual features and therefore we do not provide any programming constructs. Instead we discuss the relevant issues to be considered in each of the architectural components required to enhance the design and development of mobile agent systems. We do not claim that this model is complete, and recognize the fact that it will only reach such a state after several components have been programmed, implemented and intensively tested. However, in the meanwhile, there are benefits associated with this preliminary model, namely that the model provides a clear understanding of the architectural issues in mobile agent computing, giving novice researchers and practitioners who enter the field for the first time a foundation for making sensible decisions when researching, designing and developing mobile agents. The model is also significant in that it provides a benchmark for researchers and developers to measure the capabilities of mobile agents created by commercially available tool kits.

Section 2 reviews several mobile agent architectures in order to establish a base of criteria for our architectural model, which is presented and discussed in Section 3. Future research is discussed and conclusions are drawn in Section 4.

2. COMMON MOBILE AGENT ARCHITECTURES

In the first part of this section we describe available architectural standards for mobile agent systems from whence relevant architectural components are extracted. In the second part, a number of publicly available mobile agent systems are discussed in order to discover additional architectural components.

2.1 Mobile Agent Architectural standards

1.2.1 The FIPA Standard

The Foundation For Intelligent Physical Agents *(FIPA) specifications* [2000] represent a collection of standards which are intended to promote the interoperation of heterogeneous agents and the services they represent. These specifications are focussed on agent communication languages, agent services and supporting management ontologies for agent systems in general. No specific emphasis is placed on mobile agent systems and hence agent mobility and many other features specific to mobile agents are excluded from this standard.

1.2.2 The MASIF standard

The *OMG Mobile Agent System Interoperability Facility (MASIF)* [2000] is a standard specifically aimed at addressing interfaces between mobile agent systems. It presents a set of definitions and interfaces that provide an interoperable interface for mobile agent systems. The purpose of this standard is to promote interoperability between agent systems. MASIF uses two primary interfaces to achieve this purpose, namely, the *MAFAgentSystem* and the *MAFFinder* interface. These two interfaces address the four interoperability concerns MASIF identified, namely (1) a standard way of managing agents, including operations such as creation, suspension, resumption and termination; (2) a common mobility infrastructure enabling different mobile agent systems to communicate with and visit other agent systems, (3) a standardised syntax and semantics for naming services of agents and agent systems; and (4) a standardised location syntax for finding agents. The intention of the MAFAgentSystem is to address the first two concerns, while the MAFFinder interface has to realise the last two. Although the MASIF standard goes some way towards providing a standard for mobile agent systems, it excludes a number of important architectural components in its standardisation attempts. Its claim to interoperability is not as penetrating as desired since it only addresses interoperability between agent systems written in JAVA. MASIF justifies this by assuming that Java is becoming the de facto standard, which means that all mobile agent systems will be written in Java soon. MASIF does not address local agent operations such as

agent interpretation and execution. It also excludes inter-agent communication from its standardization efforts, and does not address conversion issues as such as these are too complex [Miljojic et al. 1998]. Assuming mobile agent systems to be written in JAVA, the MASIF standard uses built-in JAVA constructs, as well as CORBA services to address security and further expects the communication infrastructure to honour certain security concerns. A brief depiction of mobile agent aspects that are addressed by MASIF follows.

MASIF identifies an *authority* as the person or organisation for whom an agent acts and uses the term *agent system* as the platform on a host where agents are created, interpreted, executed, transferred and terminated. Such an agent system is uniquely identified by its name and address, and each host can contain many agent systems. Within each agent system, there can be one or more *places*, a place being the execution environment, within a specific context, where a mobile agent executes. An authority can own many agent systems, which may not all be of the same type. Agent systems belonging to the same authority are known as a *region*, and this is regarded as a security domain as it creates a trusted environment. The MAFAgentSystem interface addresses mobility as follows: to process a mobile agent's request to move, the source agent system halts the agent's thread, identifies its state, serialises the agent and its state, encodes the serialised data according to the underlying transport protocol and provides authentication information to the server before transferring the agent. On the receiver host side, the sender is authenticated, the serialised data is decoded and then deserialised, the agent is instantiated and its state is restored before execution is resumed.

MASIF's naming and locating services are based upon and use corresponding CORBA services. The *MAFFinder* interface defines mechanisms to register, unregister and locate agents, agent systems and places. It also provides a set of techniques which an authority can use to locate agents, including (1) *brute force*, where the agent is found by searching for it in every agent system in the region; (2) *logging*, where the agent is followed by its trail, indicated by leaving its next destination at each server it visits; (3) *agent registration*, where every agent registers its current location in a database that keeps the latest information about agents' locations; and (4) *agent advertisement*, where only places are registered, and an agent's location is registered only when the agent advertises itself.

It seems as though further development of the MASIF standard has either been placed on hold or has been discontinued. Not only is there no mention of this standard on the official OMG site, but references to future developments on the official MASIF site also lead nowhere. Furthermore, hyperlinks to references found in Miljojic et al.[1998] have also been discontinued.

2.2 Mobile Agent Implementations

We have selected four of the most popular mobile agent platforms to include in this discussion.

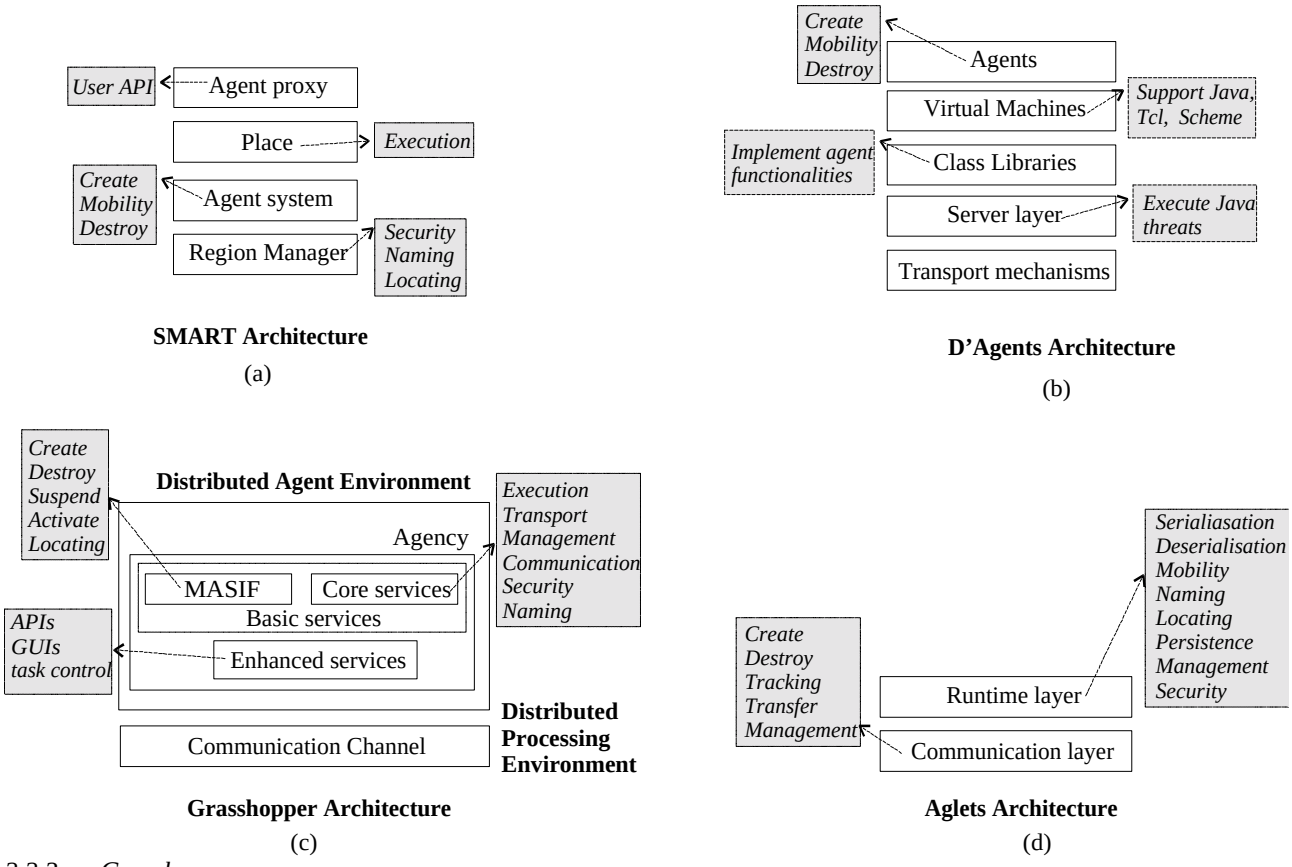
2.2.1 Scalable Mobile and Reliable Technology (SMART)

Wong et al.[2001] developed a MAF[1] compliant mobile agent platform, called Scalable Mobile and Reliable Technology (SMART). This architecture consists of four layers built on a Java virtual machine (JVM). Figure 1(a) is a simplified depiction of SMART. At the lowest layer the *region administrator* manages and enforces security policies on a set of agent systems. The *region administrator* uses a *finder module* to provide naming services to the region administrator and also to the layers above. In the second-lowest layer, the *agent system* layer, mobile agents can create, migrate and destroy themselves. The third layer from the bottom forms the execution environment and contains one or more *places*, which may exist for different execution contexts. At the topmost layer, the *agent proxy* provides the mobile agent API for applications written in SMART.

2.2.2 D'Agents

Gray et al.[2002] describe the D'Agents (formerly known as Agent Tcl) mobile-agent system which was developed to support distributed information retrieval and to characterize the mobile agent performance space. The D'Agents' architecture has five levels. At the bottom level, TCP/IP is used to provide transport mechanisms. The second layer defines a *server layer* to be implemented on all hosts to accept mobile agents. The server is a multi-threaded process and executes multiple agents as threads inside a single process, while each agent is executed inside its own (Unix) process. The next layer holds shared C++ libraries that implement agent functionality. The fourth layer provides the execution environment for each of the three supported languages (Java, Tcl and Scheme). Each such execution environment includes the interpreter/virtual machine for the supported language, stub routines allowing the agent to invoke functions in the C++ library, a state-capture module to support mobility, and a security model to enforce resource limitations. The agents themselves are defined on the top layer. Figure 1(b) is a simplified depiction of the D'Agents' architecture.

[1]The common Mobile Agent Facility (MAF) is the predecessor of MASIF.



2.2.3 Grasshopper

Grasshopper is a MASIF-compliant mobile agent platform supporting the development and execution of mobile agents.

Figure 1. *Architectural models of different mobile agent platforms*

The Grasshopper architecture [Pavlou 2000] distinguishes between two layers. The *distributed agent environment* resides on the top layer, while the bottom layer consists of the *distributed processing environment.* A host in the distributed agent environment typically includes an *agency* that has access to basic services as well as advanced/extended services. The basic services include a MASIF component with the *MAFFinder* and *MAFAgentSystem* interfaces as well as a number of core services. The core services include execution, transport, management, communication, security and naming mechanisms. Extended functionality includes adapter interfaces for external hardware/software, task control functions (e.g. itinerary), and application-specific GUIs. Figure 1(c) is a simplified depiction of Grasshopper.

The *distributed processing environment* on the bottom layer is structured into regions (optional), agencies and places. Regions provide naming services and facilitate the management of distributed components. Grasshopper uses both stationary and mobile agents. The mobile agents migrate autonomously between different locations, while the stationary agents mostly reside permanently in their creation agency, offering services to other agents or entities. An agency provides the runtime environment for mobile agents which, offers the basic platform functionality that includes agent life cycle management, agent transport, communication, persistence and security. Each agency runs on its own JVM and consists of one or more places. In the Grasshopper architecture, a *place* that is residing in an agency is considered as a grouping of agents that provide the same functionality. A region facilitates the management of agencies, places and agents as well as the tracking and finding of agents. Each region therefore contains a region registry that acts as a directory service which also provides various lookup operations and search criteria.

2.2.4 Aglets

The Aglets initiative [Lange, 1997] is probably one of the best known mobile agent projects, where mobile agents are referred to as 'aglets'. The Aglets architecture consists of two layers and two APIs that define interfaces for accessing layer functions. The *runtime layer* (top layer) consists of a core framework and subcomponents to provide the following mechanisms fundamental to aglet execution: serialization and deserialization; class loading and transfer; reference management and garbage collection; persistence management; maintenance of byte code; and protecting hosts and agents (aglets) from malicious entities. These services are defined through the API on this layer. The *communication layer* (bottom layer) uses a communication API that defines methods for creating and transferring agents, tracking agents

and managing agents in an agent-system-and-protocol-independent way. The communication API is derived from the MASIF standard. The Aglets architecture uses the Agent Transfer Protocol (ATP), modelled on the HTTP protocol, as the default implementation of the communication layer.

Aglets uses the following life-cycle: At first a new aglet must be instantiated. This can be done by either creating a new instance or cloning an existing aglet. Once created, an aglet object can be dispatched to and/or retrieved from a remote server, deactivated and placed in secondary storage, then activated later. An aglet can dispatch itself to a remote server which causes the aglet to suspend its execution, serialise its internal state and byte code into the standard form and then be transported to the destination. On the receiver side, the aglet is reconstructed according to the data received from the origin, and a new thread is assigned and executed. Figure 1(d) is a simplified depiction of the Aglets architecture.

3. PROPOSED ARCHITECTURAL MODEL

Considering the issues described in MASIF, the discussed systems, as well as other systems studied [Schoeman 2003], we have identified several architectural components and subcomponents to be included in a mobile agent system. Figure 2 depicts the proposed architectural model for the design of a mobile agent system. As in other architectural models, a layered approach is followed. The layers are organised according to the proposed life-cycle of a mobile agent. Before conducting a detailed discussion on the individual layers, we describe such a life cycle.

3.1 Mobile Agent Life Cycle

At the local host (see Figure 2), an authority (the person or organisation for whom an agent acts) uses the *Authority API* to create one or more agents in the *Agency*. Such an agent moves through the *Execution* layer to the *Mobility* and *Communication* layers where standard mechanisms to ensure mobility and effective agent communication are added. Fault tolerance mechanisms are added at the *Persistence* layer. At the *Management Services* layer, the agent is serialised, before being encrypted at the *Agent Security* layer. At the *Server* layer, the agent is named and registered for future reference. The agent receives credentials for access to other hosts at the *Host Security* layer before it is encoded in a suitable transport protocol at the *Network* layer for transport through the network. On arrival at a remote host, the *Network* layer removes the transport protocol envelope and sends the agent to the *Host Security* layer where the agent seeks host access by submitting its credentials. If it is accepted, the host server registers the agent at the *Server* layer, before the agent is decrypted at the *Agent Security* layer. The *Management Services* layer is responsible for deserialising the agent and performing any conversions that may be necessary. The agent moves through the *Mobility*, *Communications* and *Persistence* layers to be executed at a *Place* in the *Execution* layer. During execution, the agent might interact with the *Persistence* layer for storing information, the *Communication* layer for communicating with other agents or entities, and the *Mobility* layer for future migration requests. Once executed, the agent follows the path downwards through the architecture in a similar fashion as at the local host.

3.2 Model layers

1.3.1 Authority API Layer

We use the MASIF definition for the agent's authority to be the person or organization for which the agent acts. The authority API refers to the application interface that enables an authority to interact with and manage its mobile agents both locally and remotely, including tasks such as creating agents, communicating with them, and destroying them. Da Silva et al. [2000] remind developers that there are two interface types to design between an authority and its agency, namely an interface for from-authority-to-agents, as well as an interface covering from-agents-to-authority interaction. The first type of interface solves most of the interaction needs between the authority API and the agents. There are, however, also instances in which agents should take the communication initiative, such as reporting back functions, or when the agents encounter problems, in which case the second type of interface is required. Although there are many commercial agent development tool kits available, truly standardised system/independent agent APIs do not exist yet, and this necessitates further research and development.

1.3.2 Agency Layer

We use the term *agency* instead of *agent system* to refer to the environment on a host where agents are created and terminated through an authority API. The *agent system* refers to the entire mobile agent platform stretching from the lowest to the highest level. To simplify agent design, we suggest the use of Wang et al's. [2000] categorisation of agents into three basic types of agents, namely *system agents*, *participation agents* and *user agents*. The system agents take responsibility for administering the agency as well as other system functions residing on other layers of the architecture. These system agents can be further classified into *manager agents*, performing management tasks; *facilitator agents*, facilitating communication; *monitor agents*, logging events; *repository agents*, responsible for retrieving and adding information as well as querying repositories; and *interface agents*, providing the necessary API as well as interfacing with other entities and applications. Participation agents provide system support for cooperative processes in execution

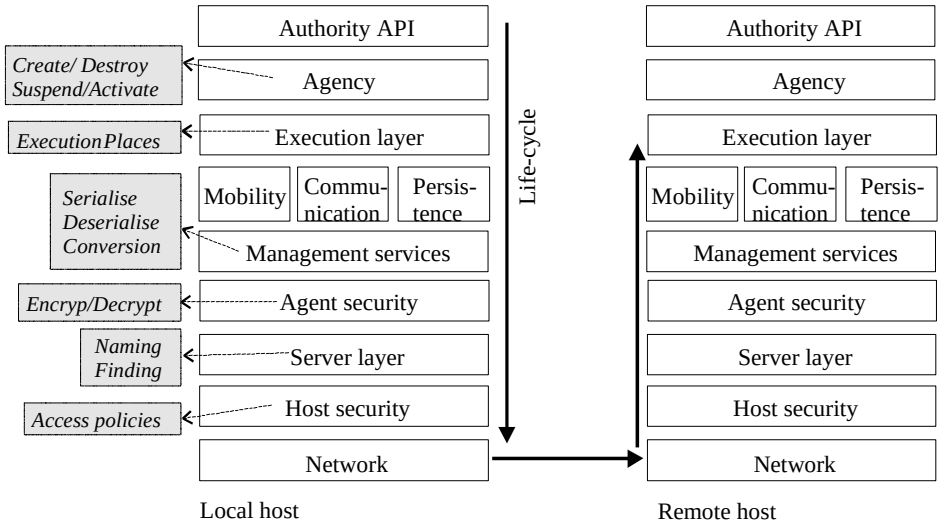


Figure 2. *Proposed architectural model for mobile agent development*

places. These agents can be further classified into subclasses to simplify agent design, but the classification depends on the type of mobile agent application. This implies that a generic classification for this type of agent will not be sensible. For example, in an e-commerce environment one can create *negotiation* and *mediation agents*, but these classes of agents will not make much sense in an information retrieval application. User agents are those agents created by an authority for a specific purpose to act on behalf of its owner. From the description of the different types and classes of agents, it is clear that both system and participation agents are created as part of the agent system, since they support the operations of the entire system, whilst the user agents are created afterwards by the authority to perform a specific user task. Many programming patterns have been described to create and maintain these agents. For more information see Aridor & Lange [1998], Kendal et al. [2000], Singh et al. [2002], Silva & Delgado [1998] and Tahara et al. [1999].

1.3.3 Execution layer

We use MASIF's description of the execution environment namely, to define a *place* as a context in which an agent executes. According to Da Silva et al.[2000], a place has two main objectives, namely to provide a conceptual and programming metaphor where agents are executed and meet other agents, and to provide a consistent way to define and control access levels, and to control computational resources. Agent places can also have CORBA-support, facilitating external applications to communicate with the agents.

The execution environment is typically responsible for hosting the interpreter/virtual machine for the supported agent language, hosting stub routines allowing the agent to invoke different library functions, granting access to local resources in a selective manner, binding operating systems functions, binding to the network layer and providing an environment where the agent can interact with or query its environment. Although the agent may be executed in either machine or interpreted language, it is often preferable to express the agent in interpreted language since this supports heterogeneity better, and has the advantage of late binding as well as the improved potential to add security mechanisms [Harrison et al. 1995]. One of the major criticisms of mobile agents is that their execution environment can easily be exploited by a guest agent [Dale 1997]. *Constrained execution* is therefore quite important to control granted access to host resources and thus maintain the integrity of, and confidence, in the host site. Take note that constrained execution in this context is not intended for security purposes. Access rights to the host (different to access to host resources) are addressed on the Host Security layer. The notion of places in the execution environment addresses this concern to a certain extent, as it is easier to allocate access limitations on a place than to restrict the entire execution environment where the access limitations imposed on a place might be different for different places/agents. These access limitations on a place can be defined in quantifiable measures such as number of CPU cycles, number of files opened, et cetera. According to Dale [1997], not only the limitations themselves, but also the agent's awareness of such limitations, will affect the manner in which an agent conducts itself, since the exhaustion of a constraint will typically result in the agent being terminated.

1.3.4 Mobility layer

Software agent systems and especially intelligent agent systems have been around for a long time, but to move these agents over the Internet takes more than the readiness of the underlying network. In fact, the mobility of many commercial mobile agent systems is restricted, being defined as an agent visiting a host, and then directly returning to its

agency. This is largely due to the relative infancy of the field (moving agents vs. stationary agents) and the lack of programming standards to facilitate mobility.

Agent mobility can either be implemented as weak or strong mobility. In weak mobility, the agent's state is represented in data structures and migration is only allowed at specific points in the agent code. In strong mobility, the agent's state is captured at the underlying thread which allows for migration at any point in the agent's execution [Triphati 2001]. With weak mobility, the agent will restart on the new host with its current data, while strong mobility allows the agent to continue execution from the point in its instructions where it was transferred [Horvat 2000]. Most Java-based mobile agent systems supports weak mobility, while systems not based on Java often provide strong mobility [Horvat 2000, Picco 2001, Tripathi 2001]. Systems with weak mobility vary in the way agents request to move. Some use a simple 'go' statement causing the agent code and data to be moved to a new host, where a predetermined procedure or method is invoked. Others associate an itinerary listing the succession of hosts to visit, with each agent, and the method to invoke on each host. Some systems invoke the same method at each stop, while others allow the agent to specify different methods for each stop.

Mobility often necessitates the transport of agent classes. Whether all classes required by the agent code should be transported as part of the agent transfer protocol, or whether classes should be obtained on demand from a designated code-base server during execution, is an area where the purpose and circumstances under which the application will be used, have to be taken into account. Class transfers are usually approached in two ways: (1) all classes required by the agent code can be transported as part of the agent transfer protocol, or (2) classes can be obtained on demand from a designated code-base server during execution [Gray et al. 2002, Milojicic 1999, Triphathi et al. 2001]. D'Agents use the first approach, which actually makes agent transfer heavyweight, since more classes than the agent actually needs, may be transferred and it has the advantage that the agent needs no further communication with its previous host for code transfer [Triphathi et al. 2001]. SMART uses the second approach, through the process of dynamic aggregation, which allows the agent to attach new code and data at runtime, reducing the amount of code transferred with the agent. It also reduces bandwidth requirements and speeds up the packing process before transmitting an agent to another host [Wong et al 2001], but slows down agent execution [Triphathi et al. 2001] and is not suitable for disconnected operations. An alternative approach is sometimes used where a thirdparty could be asked to serve as code cache to allow the sending host to disconnect, or code could be cached at each host [Gray et al. 2002]. A number of variations on these approaches are discussed by Gray et al. [2002], Miljojic et al. [1998] and Triphathi et al. [2001].

1.3.5 Communication layer

Two of the critical characteristics of mobile agents are their capabilities to collaborate and to share information. Without interoperability it will be impossible to realise the potential capabilities and benefits of the mobile agent technology. According to Pinsdorf [2002] mobile agent systems are interoperable if a mobile agent from one system can migrate to the second system, the agent can interact and communicate with other agents (local or even remote agents), the agent can leave this system and it can resume its execution on the next interoperable system. Interoperability aspects are often impeded by complexity and security concerns. We do not support MASIF‘s claim that interoperability issues can be restricted to mobile agents written in the same language as all mobile agent systems will be written in Java in the near future. In fact, we believe that standardisation on a particular language throughout the Internet will most probably not be reached in this lifetime. Our point is strengthened by Dale [1997] who points out that the behaviour expected from the mobile agent often dictates the source programming language, since different languages have varying domains of applicability.

Mobile agents need to communicate with each other and sometimes also with other non-agent services and resources. Gray et al. [2002] point to four important communication issues to be included in the design of a mobile agent, namely (1) how to identify another party with which to communicate; (2) the required communication format; (3) how to pass data from one party to the other (messaging); and (4) how to maintain communication with a moving agent. We believe that identifying an agent or a communicating party is inherent in each of these issues and therefore should not be treated as a separate communication feature. We subsequently discuss the last three of these communication issues and refer to how the identification issue has a bearing on each of them.

Inherent in agent communication are the issues surrounding the *required communication format*. Because mobile agents operate in a heterogeneous environment, interoperability cannot be achieved if there is not a standardised means through which agents can understand and communicate with their functional environment. According to Finin et al. [1998] an agent communication language (ACL) provides the capacity to integrate disparate sources of information. An ACL is usually composed of an inner context language with a common vocabulary with ontology, an outer communication language and message passing mechanisms. JAVA and Tcl/Tk are commonly used as the inner context language. An ACL must support three important communication aspects, namely (1) how to translate between different ACLs; (2) how to guarantee the semantics of concepts, objects and relationships; and (3) how to share the intended messages to be communicated.

There are many examples of standardisation efforts with regards to an ACL. We briefly mention two examples. The *Knowledge Sharing Effort* [Genesereth & Finin 1992, Neches et al.1991] developed three languages to address the abovementioned ACL issues, including the *Knowledge Interchange Format (KIF)*, which was specifically designed to address the translation aspects of an ACL, *Ontolingua* to address the semantics issue, and the *Knowledge Query and Manipulation Language (KQML)* to address the issue of sharing messages in inter-agent communication. In another example, one of the primary activities of the FIPA standard is to design a standard modelling language to support agent software engineering and to guarantee a high quality development process to construct multi-agent systems [Calisti 2003].

Messaging plays an important role in inter-agent and agent-host communication. Aridor and Oshima [1998] discuss two basic methods for the delivery of messages including (1) *Locate-and-Transfer*, where the agent is located with the aid of the naming services and then the message is transferred directly to it, i.e. two distinct phases exist during message delivery, and (2) *Forwarding*, where locating the receiving agent and delivering the message occurs in a single phase. The first may be problematic, since the agent may already have migrated when the second stage of transfer takes place. The latter may be more efficient for smaller messages, while a large message can be delivered more efficiently through the former method.

Communication maintenance approaches to address agent communication issues can be divided into three categories namely synchronous communication (such as offered by TCP), asynchronous communication (IP, RMI or RPC) and indirect communication (event-notification and shared group/meeting objects) [Horvat 2000, Tripathi et al.2001]. Gray et al.[2002] concur with these categories and subdivide synchronous communication into (1) communication where messages containing strings or arbitrary data are passed and (2) communication where messages containing serialised objects are passed. In all categories agents need to know each other's names to establish or maintain communication.

An example of why it is important to distinguish between the different communication categories in mobile agent systems is illustrated in the following example. In synchronous communication, mechanisms must be included to cater for a participating agent to migrate since migration of the agent will disrupt the communication session. This will not happen in asynchronous communication, for if the agent is transferred, an exception will be thrown, and the initiating agent can simply re-execute the lookup and binding protocol to re-establish communication with the other agent at its new location.

Most mobile agent systems use more than one of these mechanisms, as well as broadcasting/multicasting when sending the same message to multiple receivers [Horvat et al. 2000]. D'Agents, for example, uses all three by keeping a simple directory service, implemented as a collection of stationary cooperating agents [Gray et al. 2002]. Telescript agents [White 1994] typically 'meet' other local agents and then invoke methods on objects in the other agents (example of asynchronous communication), but these agents can also communicate by sending events (an example of indirect communication).

Maintaining security during communication, especially while providing remote communication to visiting agents, is an important factor to consider during the design of a mobile agent system. Security concerns are addressed on the host and agent security layers.

1.3.6 Persistence layer

According to Harrison et al. [1995], the concept of an agent migrating between hosts implies that the user is not reliant on the system launching the agent and is not (or should not be) affected if a host fails, therefore persistence is built into the notion of mobile agents. However, explicit persistence mechanisms should be integrated into a mobile agent system, to avoid loops and interminable waiting upon agents to return to their agent system. Furthermore, since mobile agents do not need to maintain a permanent connection, their state is centralised within themselves, implying that an improved fault-tolerance can be expected from this technology. However, specific mechanisms have to be embedded into mobile agent systems to handle fault situations, such as breakdown of connections or hosts, destruction of the agent, or network errors causing the agent to get lost. While most systems offer little support for failure detection and recovery [Picco 2001, Tripathi 2001], a number of systems provide some form of persistence and fault-tolerance for mobile agents by means of a *checkpoint-restore mechanism* to restart agents [Gray 2002, Horvat 2001, Picco 2001]. According to this mechanism the agent's state information is checked before and after execution on a host server, and when the server is restarted, a recovery process restarts any agents left on the server at last shutdown [Horvat 2001]. The Tacoma architecture [Johansen 1995] achieves persistence by allowing agents to create folders inside cabinets (storage space on disk); on boot-up, Tacoma then examines the 'system' cabinet and launches new agents for all the folders detected there. Other persistence mechanisms are discussed in Triphati et al. [2001] and Vogler et al. [1997].

1.3.7 Management Services layer

Serialization, deserialization and conversions, where necessary, are done on the *Management Services* layer of the proposed architectural model. In essence *object serialization* in mobile agent technology is the ability to read and write

objects to byte streams commonly for saving session state information and for sending objects (agents and their states) over the network. Serialization adds lightweight persistence as well as limited security to the agent as it protects private and transient data. Although object serialization in general does not contain any encryption/decryption algorithms in itself, it writes to and reads from streams, so it is possible for it to be coupled with any available encryption technology. Java includes serialization through an API that allows the serialised data of an object to be specified independently of the fields of the class and allows those serialised data fields to be written to and read from the stream using the existing protocol to ensure compatiblity with the default writing and reading mechanisms. Aglet hosts use the standard *Java ObjectSerialization* mechanism to export the agent state as well as the aglet itself to a stream of bytes. However, the state of the execution stacks and program counters of the threads owned by the aglet are not serialised, which implies that when an aglet is dispatched, cloned, or deactivated, any relevant state sitting on any stack of a running aglet, as well as the current program counter for any thread, is lost [Venners 1997]. Grasshopper uses the MASIF serialization/deserialization process also based on the JAVA language. Examples of other methods of serialization (supporting other languages) are discussed in the Waterken Web [2003], Le Goff et al. [2001] and Procopio [2002].

1.3.8 Agent Security

A malicious hosting node can launch several types of security attacks on a mobile agent and divert its intended execution towards a malicious goal or alter its data or other information in order to benefit from the agent's mission [Sander & Tschudin 1998]. Bierman & Cloete [2002] describe four classes of security threats being imposed on mobile agents by malicious hosts, namely (1) *integrity attacks*, which include integrity interference and information modification; (2) *availability refusals*, which include denial of service, delay of service and transmission refusal; (3) *confidentiality attacks* including eavesdropping, theft and reverse engineering, and (4) *authentication risks*, which include masquerading and cloning. In the same paper, three types of counter-measures were presented to address these classes of problems. The first type of counter-measure refers to *trust-based computing* where a trusted network environment is created in which a mobile agent roams freely and fearlessly without being threatened by a possible malicious host. A trusted environment can be achieved through tamper-resistant hardware, an agreement between specific hosts, and inclusion of detection objects in the mobile code. However, a trusted environment also goes against the notion of mobility, thus having access to unlimited Internet information. A second type of counter-measure that can be considered includes methods of recording and tracking that make use of the itinerary information of a mobile agent, either by manipulating the migration history or by keeping it hidden. Examples of specific recording and tracking methods include itinerary recording and tracking mechanisms [Roth 1998, Schneider 1997], server replication [Misky et al. 1996], et cetera. The third type of counter-measure includes cryptographic techniques that utilise encryption/decryption algorithms, private and public keys, digital signatures, digital timestamps, and hash functions to address different threat aspects. Examples include cryptographic tracing [Vigna 1997], encoding with encrypted functions [Sander & Tschudin 1998], partial result encapsulation [Jansen 2000], et cetera. Many of the abovementioned methods are theoretical at this stage and have not yet been implemented, due their complexity. As the malicious host problem is not simple, much research is currently being conducted to provide better detection and prevention mechanisms.

1.3.9 Server Layer

A global *naming scheme* managing agent names is important for identifying, controlling and locating agents [Miljojic 1998]. Moreover, a mobile agent system requires a *name service* to locate resources, specify agent servers for migration and also to establish inter-agent communication [Triphati 2001]. MASIF uses an agent's name (as assigned by its authority), its identity (a unique value within the scope of the authority) and the agent's system type (e.g. Aglet) to form a globally unique name for each agent. As described earlier, MASIF uses four basic techniques to find an agent, including brute force, logging, agent registration and agent advertisement. Other systems also use these, a hybrid of these, as well as additional methods to locate. For example, Aridor & Oshima [1998] use the brute force search in parallel or in sequence, as well as a database for agent registration as a predefined directory server used to register, unregister or locate agents. Other agents can then use this directory to find the agent. Since communicating agents need to agree in advance on the naming server, an architecture where agent servers share a default naming server simplifies the registration system.

1.3.10 Host Security

A host is faced with two potential threats from mobile agents, namely (1) a malicious agent, which might be a virus or Trojan horse vandalising the host, or (2) a benign agent that might simply abuse host's local resources. In an uncontrolled environment, mobile agents can potentially run indefinitely and consume the system level resources such as files, disk storage, I/O devices, etc., in their execution environment. Therefore resource consumption, such as CPU time, disk storage, number of threads, number of windows, network bandwidth and the like, should be limited and access

control policies that define accessibility to these resources must be put in place [Feridun & Krause 2001, Gray et al 2002, Picco 2001, Tripathi 2001].

In both instances, three basic types of counter-measures can be applied, i.e. *authentication*, *verification* and *authorisation*. Verification mechanisms address the malicious agent threat and require the checking of code to make sure that it does not perform any prohibited actions. Protection mechanisms against a potentially malicious host can of course prevent such verification procedures from being implemented, which means that the verification mechanisms can only be confined to the execution environment and hence only manage the agent during execution, making sure that the agent does not try to corrupt the execution environment.

The XML-based Security Assertion Markup Language [OASIS 2003] is a recent standardisation effort to address the *authentication* and *authorisation* concerns. One of the major design goals with SAML is single sign-on, which implies the ability of a user to authenticate in one domain and use resources in other domains without re-authenticating. Although the OASIS standard is primarily aimed at Web services that allow the exchange of authentication and authorization information among business partners, it is of particular importance for mobile agents as they engage and interact with resources and other agents in various domains. With it, a security administrator can express advanced security requirements, such as time- or event-based restrictions.

1.3.11 Network Layer

The network layer is responsible for the final encoding of the encrypted serialised agent object so that it can be transported by the underlying network to its next host. Although there are several ways to perform this encoding, the IBM Aglet's workbench team proposed an Agent Transfer Protocol (ATP) as a mobile agent standard to be adopted for transporting mobile agents. ATP is a platform-independent, application-level protocol for distributed agent-based systems for the purpose of transferring mobile agents between networked computers. Some other mobile agent systems simply use the underlying HTTP or TCP/IP protocols as transport protocol, for example Aridor & Oshima [1998] and D'agents [Gray et al. 2002].

Security mechanisms can also be included in the agent's transport protocols. For example, protocols like Secure Socket Layer (SSL) and Transport Layer Security (TLS), although a bit heavyweight, can be used for securing transmission of data between two hosts. Alternatively, the Key Exchange Protocol (KEP) offers a lightweight transport security mechanism, which suits the notion of small transferable objects better.

4. SUMMARY & CONCLUSIONS

Reviewing the publications on mobile agents, it is obvious that there is much research interest in the topic. The many (rather recent) Internet services, applications and technologies (e.g. Web Services and XML) are playing a major role in the further exploration of mobile agents as a viable Internet technology. The standardisation of these services, applications and technologies resulted in their quick adoption, especially when considering the age of some of them. The mobile agent platform has the potential to use these services, applications and technologies to enhance the capabilities of the Web tremendously, but has, to date, sadly failed to make a real impression. It is our belief that the underlying reason for this is the lack of architectural and open source standards for this technology. In this paper, we addressed this problem and proposed an architectural model that represents the different components and design aspects of mobile agent systems as a first step on the way to establishing a standard that can be used during the design and development of mobile agent systems. The proposed model follows a layered approach, enabling developers to focus their attention on issues on a specific layer that forms a functional unit. This model has to be taken further by researchers and designers to design and develop open source patterns addressing the different aspects of mobile agent systems.

5. REFERENCES

ARIDOR, Y. AND LANGE, D.B. 1998. Agent Design Patterns: Elements of Agent Aplication Design. In *Proceedings of the Second International Conference on Autonomous Agents*. Minneapolis/St. Paul, USA. 108 - 115.

ARIDOR, Y. AND OSHIMA, M. 1998. Infrastructure for Mobile Agents: Requirements and Design. In *Proceedings of the Second International Workshop on Mobile Agents (MA'98)*. Stuttgart, Germany. 38-49.

BERNERS-LEE T., HENDLER J. AND LASSILA, O. 2001. The Semantic Web. The Scientific American.com. http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21.

CALISTI,. M. 2003. FIPA standards for promoting interoperability of industrial agent systems. FIPA Presentation, Agentcities, Information Days. Barcelona, Spain.

DALE, J. A Mobile Agent Architecture for Distributed Information Management.1997, Ph.D. thesis, University of Southampton.

DA SILVA, A.R., DA SILVA, M.M. AND ROMµO, A. 2000. Web-based Agent Applications: User Interfaces and Mobile Agents. http://citeseer.nj.nec.com/499980.html

FIPA: THE FOUNDATION FOR INTELLIGENT AGENTS. 2000. http://www.fipa.org/specifications.

FERIDUN, M. AND KRAUSE, J. 2001. A framework for distributed management with mobile components. *Computer networks.* Vol 35. 25 - 38.

FININ, T., LABROU, Y., AND PENG, Y. 1998. Mobile Agents Can Benefit from Standards Efforts on Interagent Communication. *IEEE Communications Magazine*. http://www.cs.umbc.edu/~finin//papers/IEEECommDraft.pdf

GENESERETH, M.R. AND FININ, R.E. Knowledge Interchange Format, Version 3.0 Reference Manual. Computer Science Department. Stanford University. 1992. http://logic.stanford.edu/kif/Hypertext/kif-manual.html

GRAY, R.S., CYBENKO, G., KOTZ, D., PETERSON, R.A. AND RUS, D. 2002. D'Agents: Applications and performance of a mobile agent system. *Software: Practice and Experience.* Vol 35. Number 6. 543 - 573.

HARRISON, C. G., CHESS, D. M. AND KERSHENBAUM, A. 1995. Mobile Agents: Are they a good idea? IBM Research Report. IBM T. J. Watson Research Center.

HORVAT, H., CVETKOVI, D., MILUTINOVI, D., KOVI, P. & KOVAEVI, V. 2000. Mobile Agents and Java Mobile Agent Toolkits. In *Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS-33).* CD-ROM. p. 10. IEEE Computing Society, Los Alamos, CA.

JANSEN, W.A. 2000. Countermeasures for Mobile Agent Security. *Computer Communications*. Special issue on advanced security techniques for network protection. Elsevier Science.

JOHANSEN, D., VAN RENESSE, R., AND SCHNEIDER, F.B. 1995. An Introduction to the TACOMA Distributed System. Technical Report. 95-23. Department of Computer Science. Institute of Mathematical and Physical Sciences.

KAGAL, L., PERICH, F., CHEN, H., TOLIA, S., ZOU, Y., FININ, T., JOSHI, A., PENG, Y., COST, R.S. AND NICHOLAS, C. Accessed May 2003. Agents making sense of the Semantic Web. NEC Research Institute CiteSeer. http://citeseer.nj.nec.com/531295.html.

KENDALL, E.A., KRISHNA, P.V., SURESH C.B. AND PATHAK, C.V. 2000. An Application Framework for Intellgent and Mobile Agents. *ACM Computing Surveys*. Vol 32. No 1.

KOTZ, D. AND GRAY, R.S. 1999. Mobile Agents and the Future of the Internet. http://www.cs.dartmouth.edu/~dfk/papers/kotz:future2.

KOTZ, D., GRAY, R. AND RUS, D. 2002. Future Directions for Mobile Agent Research. IEEE Distributed Systems. http://dsonline.computer.org/0208/f/kot.htm.

LANGE, D.B. 1997. Java Aglet Application Programming Interface (J-AAPI) White Paper - Draft 2. http://www.trl.ibm.co.jp/aglets.

LE GOFF, H. STOCKINGER, J.-M., WILLERS, I., MCCLATCHEY, R., KOVACS, Z., MARTIN, P., BHATTI, N. AND HASSAN, W. 2001. Object Serialization and Deserialization Using XML. The Compact Muon Solenoid Experiment.CMS CERN, CH-1211 GENEVA 23, Switzerland. Report No: CMS NOTE 2001/025. http://arxiv.org/ftp/physics/papers/0105/0105083.pdf.

MASIF: The OMG Mobile Agent System Interoperability Facility. 2000. Lecture Notes in Computer Science. Springer-Verlag: SpringerLink. Berlin, Germany. LNCS 1477, p. 50 ff. http://link.springer.de/link/service/series/0558/bibs/1477/14770050.htm.

MILOJICIC, D. 1999. Trend Wars Mobile Agent Applications. *IEEE Concurrency*. Vol. 7. No. 3. 80 - 90.

MILOJICIC, D., BREUGST, M., BUSSE, I., CAMPBELL, J., COVACI, S., FRIEDMAN, B., KOSAKA, K., LANGE, D., ONO, K., OSHIMA, M., THAM, C., VIRDHAGRISWARAN, S. AND WHITE, J. 1998. MASIF: The OMG Mobile Agent System Interoperability Facility. *Personal Technologies.* Vol 2, No 2. 117 - 128.

MINSKY, Y., VAN RENESSE, R. SHEIDER, F., STOLLER, S. 1996. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*. Connemara, Ireland 109-114.

NECHES, R., FIKES, R., FININ, R.E., GRUBER, R., PATIL, R., SENATOR, T., AND SWARTOUT W.R. 1991. Enabling Technology for Knowledge Sharing. *AI Magazine*. vol 12. 1001.36-56. http://www.isi.edu/isd/KRSharing/vision/AIMag-small.html.

ORDILLE, J.J. 1996. When agents roam, who can you trust? In Proceedings of the First Conference on Emerging Technologies and Applications in Communications. Portland, Oregon.

ORGANIZATION FOR ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS (OASIS). 2003. Assertions and Protocol for the OASIS Security Assertion Markup Language 3 (SAML) V1.14. http://www.oasis-open.org/committees/download.php/1894/sstc-saml-core-1.1-draft-10.pdf.

PAVLOU, G. 2000. The Grasshopper Mobile Agent Platform. Appearing in Mobile intelligent agents for managing the information infrastructure. http://www.ee.surrey.ac.uk/CCSR/ACTS/Miami/grasshopper.html.

PICCO, G.P. 2001. Mobile agents: an introduction. Microprocessors and Microsystems. Vol. 2, No. 2. 65 -74.

PINSDORF, U. AND ROTH, V. 2002. Mobile Agent Interoperability Patterns and Practice. In Proceedings of Ninth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems. Sweden.

PROCOPIO, M. 2002. Object Serialization and Deserialization in C#. http://www.devhood.com/tutorials/tutorial_details.aspx?tutorial_id=448

ROTH, V. 1998. Secure Recording of Itineraries through Cooperating Agents. (In Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, Brussels, Belgium.147-154.

SANDER, T. AND TSCHUDIN, C. 1998. Protecting Mobile Agents against Malicious Hosts. *Lecture Notes in Computer Science.* Springer-Verlag No. 1419, 44-60.

SCHNEIDER, F.B. 1997. Towards Fault-Tolerant and Secure Agentry. *In Proceedings of the 11th International Workshop on Distributed Algorithms.* Saarbrücken, Germany

SCHOEMAN, M.A. 2003. Architectural guidelines for Mobile Agent Systems. Technical Report: TR-UNISA-2003-02. School of Computing. University of South Africa.

SHIH, T.K. 2001. Mobile agent evolution computing. *Information Sciences.* Vol 137.53 - 73.

SINGH, A.K., SANKAR AND R., JAMWAL V. 2002. Design Patterns for Mobile Agent Applications. Workshop on Ubiquitous Agents on embedded, wearable, and mobile devices held in conjunction with the Conference on Autonomous Agents & Multiagent Systems. Bologna. http://autonomousagents.org/ubiquitousagents/papers/papers/22.pdf

SILVA, A. AND DELGADO, J. 1998. The Agent Pattern for Mobile Agent Systems. European Conference on Pattern Languages of Programming and Computing. Bad Irsee, Germany.

TAHARA, Y., OHSUGA, A. AND HONIDEN, S. 1999. Agent System Development Method Based on Agent Patterns. In *Proceedings of the 21th International Conference on Software Engineering*. Los Angeles, CA, USA.356 - 367.

TRIPATHI, A.R., AHMED, T. & KARNIK, N.M. 2001. Experiences and future challenges in mobile agent programming. *Microprocessors and Microsystems.* Vol 25. 121-129.

VENNERS B. 1997. Under the hood: The architecture of aglets. http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood_p.html

VIGNA, G. 1997. Protecting Mobile Agents through Tracing. In *Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems.* Jyväskylä, Finland.

VOGLER, H., KUNKELMANN, T. AND MOSCHGATH, M.L. 1997. Distributed Transaction Processing as a Reliability Concept for Mobile Agents. In *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '97).*

WANG, A.I., HANSSEN, A.A., NYMOEN, B.S. 2000. Design Principles for A Mobile, Multi-agent Architecture for Cooperative Software Engineering

WATERKEN™ WEB. Object Serialization Specification. 2003. http://www.waterken.com/dev/Web/Object/

WHITE, J. E. 1994. Telescript Technology: The Foundation for the Electronic Marketplace. *White paper*. General Magic, Inc., Mountain View, CA, USA.

WONG, J., HELMER, G., NAGANATHAN, V., POLAVARAPU, S., HONOVAR, V. AND MILLER, L. 2001. SMART mobile agent facility. *The Journal of Systems and Software.* Vol 56. 9 - 22.