RZ 3152 (# 93198) 06/28/99
Computer Science/Mathematics 18 pages

Research Report

State of the Art of Mobile Agent Computing – Security, Fault Tolerance, and Transaction Support

Stefan Pleisch*

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

*Affiliated also to the Operating Systems Laboratory, EPFL, Lausanne, Switzerland

State of the Art of Mobile Agent Computing – Security, Fault Tolerance, and Transaction Support

Stefan Pleisch*

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

Agent technology is a field of considerable active research. Various classes of agents (e.g. intelligent agents, mobile agents) have emerged so far. The scope of this discussion is limited to mobile agents, which constitute the base technology of most agent types. This paper summarizes the current state of the art. It focuses on security, fault tolerance, and transaction support. These aspects are crucial for the further development of mobile agent technology in various application domains, such as e-commerce and network and systems management.

*Affiliated also to the Operating Systems Laboratory, EPFL, Lausanne, Switzerland

1 Introduction

In recent years, agent technology has been subject of growing research efforts. These efforts have led to the definition of various classes of agents (e.g. intelligent agents, mobile agents). We believe that mobile agents are the base technology for most of the agent classes and thus focus on these.

Mobile agents provide a valuable alternative to the traditional client/server programming model (among others), because they provide a uniform paradigm for distributed systems. Nevertheless, mobile agent technology is not yet widespread in today's applications. We believe that this is to some extent due to the lack of adequate infrastructural services such as security, fault tolerance, and transaction support, which mobile agents platforms should provide or at least support. Besides summarizing the state of the art of mobile agent systems, our survey emphasizes these crucial infrastructural services.

The idea of sending programs to and executing them at a remote site has been explored for some time. The PostScript language can be considered a rudimentary form of this because it involves sending programs to a remote processor in a printer. In 1990, General Magic launched the first commercially available mobile agent platform called Telescript[1] [Gen95]. However, it was never a commercial success, partly because it was not open to the public, and was soon abandoned. In the past few years, several mobile agent platforms such as Agent Tcl [Gra96], Aglets [IBMa], Mole [SBH97], and Voyager [Obj99] have emerged. Whereas earlier platforms were based on various scripting languages, recently emerging mobile agent platforms rely all, to our knowledge, on Java.

Standards have been defined to provide interoperability among different mobile agent platforms [Fip, MBB+98]. However, they have not yet been widely accepted. Furthermore there has been significant research centered around such mobile agent technology issues as security [KAG98, ST98, Sch97, WBS98]. Until now, however, no entirely satisfactory solutions have been proposed.

To our knowledge, no survey that centers around the same aspects as ours has been published so far. Other work surveying mobile agent technology has been published, but it either focuses on the usefulness of mobile agent technology [CHK98, Mat97], or compares different platforms [KZ97, CGPV97].

After defining the model to be used throughout this paper in Section 2, we will identify the advantages of the mobile agent paradigm in Section 3 and present some application domains for mobile agent technology in Section 4. Section 5 discusses a set of infrastructural services, including the services considered crucial for the further development of mobile agents, such as security, fault tolerance, and transaction support. The pertinent current standardization efforts are elaborated in Section 6. In Section 7 we present our conclusions.

2 Model and Definitions

Today's most widespread paradigm for distributed computing follows the client/server paradigm. In this paradigm the server is defined as a computational entity that provides some services. The client requests the execution of these services by interacting with the server. After the service is executed the result is delivered back to the client. The server therefore provides the knowledge of how to handle the request as well as the necessary resources.
The paradigm of mobile code generalizes this concept by performing changes along two orthogonal axes:

1. Where is the know-how of the service located?

2. Who provides the computational resources?

Depending on the choices made on the server and client sides, the following additional paradigms, illustrated in Table 1, can be identified [FPV98]:

[1]General Magic received a patent for its agent technology in 1997.

Remote Evaluation (REV). In the REV paradigm a component A sends instructions specifying how to perform a service to a component B. The instructions can, for instance, be expressed in Java bytecode. B then executes the request using its resources. Java Servlets[2] are an example of remote evaluation.

Code on Demand (CoD). In the CoD paradigm the same interactions take place as in REV. The difference is that component A has the resources collocated with itself but lacks the knowledge of how to access and process these resources. It gets this information from component B. As soon as A has the necessary know-how, it can start executing. Java Applets[3] fall under this paradigm.

Mobile Agent. The mobile agent paradigm is an extension of the REV paradigm. Whereas the latter focuses primarily on the transfer of code, the mobile agent paradigm involves the mobility of an entire computational entity, along with its code, the state, and potentially the resources required to solve the task. As developer-transparent capturing and transfer of the execution state requires global state models as well as functions to externalize and internalize the agent state, only few systems (e.g., Agent Tcl [Gra96], Telescript [Gen95]) support this *strong mobility* scheme. In particular, Java-based mobile agent platforms are generally unsuitable for this approach, because it is not possible to access an agent's execution stack without modifying the Java Virtual Machine.

Most systems thus settle for the *weak mobility* scheme where only the data state is transferred. Even though it does not implicitly transport the execution state of the agent, the developer can explicitly store the execution state (see below for an exact definition) of the agent in its member attributes. The values of these member attributes are transported to the next node. The responsibility for handling the execution state of an agent thereby resides with the developer.

Figure 1 compares the mobile agent paradigm with the client/server paradigm. Whereas in the latter the client calls a service on the server, the client in the mobile agent paradigm sends an agent to the server, which interacts locally and then returns to deliver the results.

As the mobile agent paradigm has the widest scope, we will focus in the following on this paradigm. The discussions however also apply to the REV and the CoD paradigm. As mentioned in the introduction, we only consider the class of mobile agents out of all possible types of agents. We can therefore use the term *agent* synonymously with *mobile agent*. Throughout this paper the following definitions (adopted from the OMG's MASIF standard [Gro97, MBB+98]) will be applied:

- *Mobile agents* are computer programs that act autonomously on behalf of a user and travel through a network of heterogeneous machines.
- *Itinerant agents* [CGH+95] move repeatedly from host to host in a network. The sequence of hosts can either be predefined or determined by the agent on the fly, based on the agent's current state and its logic.

[2]See Sun's Java servlet homepage http://java.sun.com/products/servlet

[3]See Sun's Java applet homepage http://java.sun.com/applets/index.html

	before		after	
	n_A	n_B	n_A	n_B
Client/Server	A	code,res.,B	A	code,res.,B
Remote Evaluation	code,A	res.,B	A	*code*,res.,B
Code on Demand	res.,A	code,B	*code*,res., A	B
Mobile Agent	code,A	res.		*code*,res.,*A*

Table 1: Different paradigms for code mobility [FPV98]. Code or computational entity transported between execution environments indicated by italics.

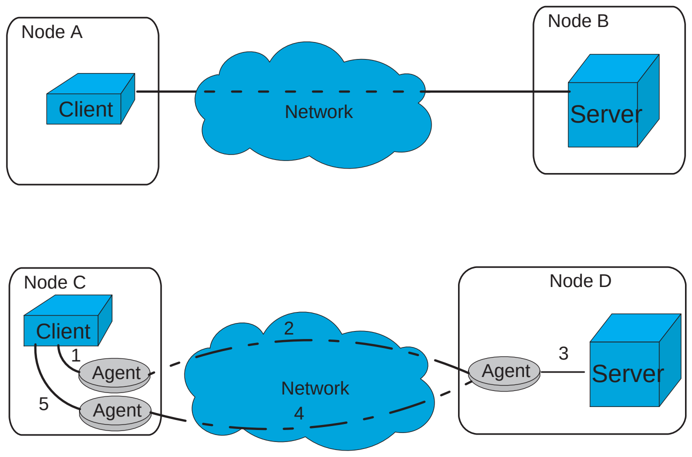

Figure 1: Comparison of the client/server paradigm (above) with the mobile agent paradigm (below)

- The *agent execution state* is its runtime state, including program counter and frame stacks, if applicable.
- We identify the *agent state* as either the execution state of the agent or the agent's attribute values that determine the course of action when the execution resumes.
- The *agent authority* is either a person or an organization on whose behalf the agent acts.
- An *agent system* is a platform that can create, interpret, execute, transfer and terminate agents. It generally corresponds to an operating system process. Like an agent, an agent system is associated with an authority for whom the agent system acts.

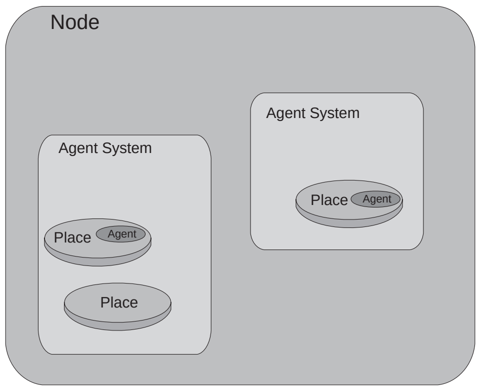

Figure 2: Relation between node, agent system and place.

- A *place* (also called a *landing pad* [JMS+98] or *computational environment* [FPV98]) is a context within the agent system (see Figure 2) that provides a uniform environment in which an agent can execute. It is associated with a particular agent system, which it partitions logically. A place provides the means for managing agents, enforcing security policies, and accessing local resources.

- The agent execution is divided into a sequence of *stages*. Two stages i and $i+1$ are separated by an agent's *move*, i.e., changing the place, or *spawn*, i.e., cloning the agent, operation.

The next section elaborates the advantages of the mobile agent paradigm over the traditional client/server paradigm.

3 Advantages of the Mobile Agent Paradigm

At the moment it is not yet clear whether mobile agent technology will establish itself as an independent paradigm for practical use in the long run.

Chess et al. [CHK98] made an attempt to estimate the benefit of mobile agent technology. Their investigation included areas such as prototyping, transactions, and scalability. They concluded that all the problems they considered can be solved using traditional client/server solutions. However, mobile agents allow a general solution to all such problems. Instead of having to create different, well-tailored solutions to every problem, mobile agents provide a generic solution to all these problems. The mobile agent paradigm has several advantages over the traditional client/server model [LO99, WPM99, CHK98]:

Communication latency and bandwidth. If the communication between two interacting entities (e.g. a client and a server) involves a considerable amount of data, it may be beneficial to move one of them close to the server instead of moving the data between them. The locality of the two entities allows them to decrease the *latency* and save *bandwidth* in the communication, especially if the mobile agents act as filters and process only the data that is really useful to the agent authority. An agent authority can build a hierarchy of filtering mobile agents where appropriate in order to reduce network traffic. Clearly the gain in latency and bandwidth must overcome the cost of sending the agent to the server node. Fuggetta et al. introduce a model for performance measurements in [FPV98]. They state that in the client/server paradigm, the overhead of the interaction depends on the size of the database accessed by the application. On the contrary, the overhead of the mobile agent paradigm is bound to the size of the code and the data sent by the application. This leads to the conclusion that the application of the mobile agent paradigm is only justified when the database has surpassed a given threshold in size.

To exploit locality it is necessary for the developer of mobile agent applications to associate the notion of a locality with every server. Based on the locality information the agent may decide to move to the server location instead of invoking the server functions remotely. This is a contradiction of the general programming model in distributed systems. In such a model the client does not have to know where the server is located. This is taken care of by the communication middleware (e.g. CORBA [OMG98a], Tuxedo [BEA96]), which transparently forwards the message to the node a server is currently running on.

Off-line processing. In *mobile computing*, roaming devices such as Personal Digital Assistants (PDA) or laptops are often disconnected from the fixed network. In addition, the emission of messages from the mobile device is expensive in terms of power consumption. Connections to the fixed network may also incur considerable financial costs. This presents an opportunity for mobile agent technology because the mobile device delegates an agent to act on its behalf and to perform the required actions instead of polling various servers. The next time the mobile device connects to the network it can collect the results. Such a mobile agent can be used not only to query servers but also act as a filter that preconsolidates messages sent to the mobile device.

Reaction time. Having mobile components allows an application to exploit locality. *Reaction time* is less significant for a local agent acting on behalf of a remote monitoring device than for the device itself. In addition resources may under certain circumstances only be accessible locally. Agents are a concept to provide external service in this situation.

Asynchronous behavior. Instead of interacting with a server in many RPC-type communications, a client can bundle the requests within a mobile agent. Having reached the server the agent starts interacting with the services locally. This also simplifies the recovery from communication failures, because either the entire agent, i.e., all the requests, arrive at the server or none at all. In space communication it is often appropriate, due to the communication latency, to use this type of interaction.

Dynamic adaptation. Mobile agents have the ability to adapt dynamically to changes in their environment. They can, for instance, react autonomously to balance the load in the network or move on to a replica of a current node that is failing. This behavior also provides mobile agents with robustness and a degree of fault tolerance.

Protocol encapsulation. Today's networks consist of many legacy applications. As their protocols evolve, legacy problems often occur. Mobile agents move to the remote legacy application and encapsulate its protocol. Other applications communicate with this application via the agent, using a proprietary protocol.

These advantages make mobile agents a suitable and beneficial technology for various application domains, elaborated in the following section.

4 Application Domains for Mobile Agents

Whereas the potential usefulness of the mobile agent paradigm has been widely accepted, the mobile agent technology has not yet found its way into today's more prominent applications. The following use cases may represent domains where mobile agent technology can make an impact and an important qualitative difference [LO99, CHK98]:

e-commerce. With e-commerce growing at a very high rate the need for mobile agents is also increasing. It has been proposed to use mobile agent technology to provide some of the services of e-commerce. Agents are sent out into the Internet by an interested user and gather the required information. If necessary they can collaborate with other agents, engaging in information sharing, exchanging or buying [MGM99]. Schemes are revised to create marketplaces where agents can deal and even take part in auctions. For this purpose an agent is equipped with the mechanisms to deliver payment for purchased goods or services on behalf of its authority.

Watchdog applications. An agent monitors a component, such as a device driver, an application, or a switch. It is able to react locally to a certain behavioral pattern of the monitored component. For instance a network management agent could monitor the network traffic until it detects traffic congestion. It then sends an email to the administrator or autonomously takes appropriate corrective actions.

Itinerant actions. Transactions may involve several nodes. If the nodes have to be visited sequentially rather than in parallel, an agent could be an alternative to performing a client/server type of call to every host.

Information gathering. Information gathering can be left to an agent's responsibility if multiple sources have to be considered or if the sources are not exactly known beforehead. An agent might accumulate some knowledge during its itinerary that allows it to make a decision about its further itinerary.

System configuration. Agents can provide a more flexible mechanism for system configuration. In particular, the dynamic dispatching of mobile code allows reconfiguration without shutting down the whole system. Active networks [TW96], for instance, rely on instructions sent with the communication packets. Dynamic system maintenance and software updates are also supported by this mechanism.

Parallel processing. Mobile agents can clone themselves and split up the work among the clones. This allows tasks to execute in parallel and distribute processing power among different nodes. An application based on mobile agent technology is thus also more easily scalable than applications consisting of monolithic blocks.

Before mobile agent applications begin to appear on a large scale, however, the mobile agent platforms need to provide the infra-structural services to facilitate agent development. Among them are security, management of agents, fault tolerance, and transaction support, as elaborated in the next section. In addition, aspects of agent technology have to be standardized to allow different agent systems to interoperate.

5 Infrastructure for Mobile Agents

Today's mobile agent platforms provide a variety of infrastructural services. These services include language support, security, transaction support, communication facilities, etc. In the following subsections we will present these services.

5.1 Languages

Even though some mobile agent platforms have been based on compiled languages such as C [Uni] most of the existing platforms use interpreted languages such as Tcl, Perl, or Java. Interpreted languages have the advantage that they are highly portable and that the mobile agent system can to some extent influence the execution of the agent code. In particular, system calls built dynamically during execution can be verified and illegal instructions potentially rejected. This is not possible with binary code format. Modifications to the interpreter provide the means to build additional features such as security into the mobile agent system. Some programming languages allow customization and extension of functionality without even having to modify the interpreter (e.g., Security Manager in Java).

Platform-independent programming languages such as Java bytecode are favored in heterogeneous execution environments.

Recently, the influence of Java has grown rapidly and all of today's newly created mobile agent platforms are, to our knowledge, written in this language. This is because Java is an interpreted language, it is highly portable and it provides a large set of standard libraries and tools that solve certain problems related to mobile agents such as serialization, remote method invocation, and the class-loading mechanism, which facilitates the migration of code [WPM99]. It is also straightforward to extend and modify the behavior of the virtual machine, for example using the security manager or the class loading facility.

5.2 Agent Collaboration

Agent collaboration deals with multiple agents working together to solve a certain problem that they could not solve, or not as efficiently, on their own. For instance, pairwise testing of connectivity in a network often requires two agents. One of them moves from node n_1 to n_2, while the other is transported simultaneously from n_2 to n_1, but not necessarily taking the same way across the network. The two agents have to be applied *simultaneously* in order to capture the current state of the bidirectional communication channel. If both channels are tested sequentially, the environment could already have changed significantly between the tests. Compared to simple agent communication using mechanisms such as remote method invocation, agent collaboration adds an additional level of information transfer between agents based on their knowledge about their environment. Systems consisting of multiple collaborating agents are called multi-agent systems. Such multi-agent systems are considered an important technology for enabling e-commerce. Agents could, for instance, trade with each other in an electronic marketplace. As a consequence, this kind of tasks requires an interoperation mechanism between agents from potentially different vendors. This has led to the definition of standardized *agent communication languages* (ACLs) [LFP99], the most important of them:

- the ARPA Knowledge Sharing Effort, which defines a framework for knowledge exchange called *Knowledge Query and Manipulation Language* (KQML) [LF97]. KQML focuses on an extensible set of *performatives*, or message types, which defines the permissible operations that agents may execute on each other's knowledge and goal stores. As the content of the messages was not part of the standard, the *Knowledge Interchange Format* (KIF), [4] a formal language was defined based on first-order predicate calculus for interchanging knowledge among disparate computer programs.
- FIPA ACL [Fip98a], developed by the Foundation for Intelligent Physical Agents (FIPA), is similar to KQML. It defines a set of standard *communicative acts*, essentially messages, and their meaning. However, it does not address a formalism for representing the message content. Correct interpretation of message content is left to the agents.

In addition, agents rely on an *ontology* [Fip98b], which defines their vocabulary and the indented meaning of the vocabulary. An ontology service enables the mapping from one agent's ontology to another, allowing the agents to communicate with each other.

The problem of knowledge representation and exchange still is an active field of research in artificial intelligence and is not further discussed in this paper.

5.3 Agent Management

In order for mobile agents to become commercially successful they need to be manageable and traceable. This means that once an agent has been injected into the network, the system administrator or the agent authority has to be able to manage and trace the agent. Supported operations should include the ability to suspend, restart, and kill the agent. Certain control mechanisms have to be built into either the agent itself or the agent platform that verify whether the agent still complies with the rules defined by the agent authority.

Another important task of the mobile agent platform is the detection of orphan agents. After an agent has completed its task or after its lifetime has expired, it must be deleted. Otherwise, valuable network bandwidth as well as processing power and memory may be wasted by these orphan agents [Bau97].

5.4 Security

In a client/server type of distributed system the security issues are well known and adequate solutions exist [PC97]. The clients and servers are generally grouped into an administrative domain with a set of registered users. In addition, the client is executed on the user's machine. Since they run in the same administrative domain, a user trusts a server or can at least prove that incorrect results have been received.

The security in a mobile code environment cannot rely on this trust relationship between the server and an agent because they are generally not part of the same administrative domain. In addition, the problem of protecting the agent and its results from malicious and faulty servers arises. Whereas the protection of the server in a mobile agent environment is a problem that has been mastered in

[4]KIF draft specification http://logic.stanford.edu/kif/specification.html

most cases, the protection of the agent is still open. The following subsections discuss server and agent protection separately.

5.4.1 Protecting the Server

Several protection techniques have been suggested to solve the problem of protecting the server. Among them are the *sandbox* model, *code signing*, the introduction of a *firewall*, and *proof-carrying code* [RG98].

The *sandbox model* [FM96] aims at containing mobile code in such a way that it cannot damage the local execution environment. To achieve this, the interpreter defines the security policies for the locally as well as the remotely originated code. Whereas local code is usually granted access to all resources (I/O, file system, etc.) the remotely originated code runs in the sandbox and is thus subject to higher restrictions with respect to resource access. It may, for instance, not be allowed to access system commands. This approach limits the useful functionality remote code can implement to displaying graphics, for instance, and thus is not attractive for real-world applications.

Code signing introduces an orthogonal concept to the sandbox model. It devides the sources of remote code into two classes, trusted and untrusted. Code from untrusted sources underlies more restrictions than code from trusted sources, which is allowed to be executed outside of the sandbox. In a sandbox supporting various access rights remote code executes with access restrictions based on its source.
Every trusted source signs its classes using a digital signature technique. The interpreter can thus verify the source of every class. This approach relies on the belief that trusted sources behave correctly. If a class from a trusted source modifies the list of trusted entities of the interpreter, it can open the door for classes from untrusted sources. The code signing technique is usually combined with the sandbox model.

The *firewall* approach to mobile code involves inspecting all the classes when they enter the interpreter's domain. Upon this inspection it can be decided by the firewall to pass the class to the interpreter or to reject it. It has been shown that it is theoretically not possible in a general case to determine whether remote code is malicious or not. A Java bytecode verifier is an instance of such a firewall, which ensures that the code complies with the Java language specification (e.g., a float does not become an integer).

Finally the technique *proof-carrying code (PCC)* [NL96] constructs a proof, that guarantees that the code does not violate some safety policies. This proof is delivered with the code. The destination verifies the correctness of the proof, which is easier than performing the proof itself. For instance it is possible for some programs to construct proof that they do not contain any buffer overflow. However there are properties that can never be proved using this method, for instance properties related to information flow and confidentiality.
This technique can be part of a firewall.

5.4.2 Protecting the Agent

The problem stems from the fact that the code of the agent is executed in an untrusted environment. Without protection of the agent the execution environment can alter or destroy the agent's code and the results accumulated during its itinerary. Service providers could take advantage of such possibilities of tampering.

For instance consider an agent that books the cheapest flight to New York. After visiting several airline servers it arrives at a malicious server of an airline. The service provider is aware that its price offer is not the lowest and therefore the server modifies the collected prices of other airline's offers such that its price becomes the lowest. Encrypting the results of the agent is not sufficient because in this case the execution environment simply discards some or all other offers.

Another example is an agent that is equipped with a credit card number in order to issue payments. The credit card number should be only visible to the node where the agent buys a service or goods. It is particularly difficult to ensure this privacy if the itinerary of the agent is not known beforehead.

These simple examples show that protocols are needed that provide effective protection to the agent.

A possible solution for protecting the agent is based on a *tamper-proof environment* (TPE) [WBS98]. This concept usually relies on a hardware blackbox. It provides a well-defined, restricted interface to the outside environment as well as an execution environment for agents inside the blackbox. The restricted interface does not allow one to inspect or tamper with an agent's code inside the blackbox from the outside. The agent has to trust a TPE, or rather the TPE's manufacturer, which is usually not the same company as the service provider. An agent is encrypted with the public key of the TPE and therefore is of no direct use to the service itself. It is forwarded to the TPE, which decrypts it using its private key and finally starts to execute the agent code. The interactions between the agent and the local environment or other agents are also handled by the TPE. When the stage's execution is completed the agent moves on.

The security is based on the agent owner's trust towards the TPE manufacturer. The latter is believed not to provide malicious TPEs because of its business interests. Clearly the TPE manufacturer would be out of business instantly if somebody could prove that it delivered a malicious TPE or that its TPE is not secure from tampering.

However, the agent does not have a guarantee about the correctness of the server information. Since the server is generally not in the same administrative domain, the agent cannot benefit from this additional trust level.

As the use of TPEs incurs considerable organizational and financial overhead we believe that its application will be limited to particularly security-sensitive domains (e.g. banks, stock market). To reduce the size of the TPE only the crucial parts of the agent, such as the certificates or the keys, are kept in the TPE, instead of accommodating the entire agent. Smartcards are an example of such a TPE.

Karjoth et al. [KAG98] suggest several algorithms for protecting the results of free-roaming agents. These algorithms allow to detect the tampering of the agent's results. The results are only protected up to the first malicious host. Two cooperating malicious hosts can remove the results of the places between them from the agent without being detected.

Whereas one class of algorithms is based on a *per-server digital signature* scheme, another uses *hash chains*. In the per-server digital signature scheme the result of a stage's computation is encrypted using the public key of the agent authority. Place p_i signs this expression and forwards it to p_{i+1}. In order to detect the removal of results, some random number is encrypted along with the result of the computation. In addition, a hash value over the result at p_{i-1} and the ID of p_{i+1} chains the different stages together. Changing the order of encryption and signing allows the identity of the previous places to be hidden. This scheme relies on the availability of a public-key infrastructure.

Hash chain algorithms are based on forwarding a value in addition to the encrypted result to the next place. This value can be a hash value containing the computation's result, a random number, the previous hash value to build the chain, and the ID of p_{i+1}.

Sander and Tschudin [ST98] suggest computing with *encrypted functions* to provide fully software-based protection of an agent against malicious hosts. Assume that an agent knows how to compute f and requires $f(x)$ from a service located at place p, but wants to keep f a secret. The agent therefore transforms (encrypts) f to some other function $E(f)$ that hides f and may also produce encrypted output data. $P(E(f))$ describes the program that implements $E(f)$ and is sent to p. Therefore, p only learns about $P(E(f))$, which it applies to its input data x. The result $P(E(f))(x)$ is sent back to the agent, which decrypts it and obtains $f(x)$.

Currently this approach only supports polynomial and rational functions. In addition it does not allow continuous interactions between an agent and a place. This approach is thus not well suited for general use.

Another interesting approach is to combine the security issues with fault tolerance [Sch97]. To protect the mobile agent from malicious hosts the agent execution for every stage is replicated. Instead of sending the agent from stage i to only one place of stage $i+1$, it is sent to a group G_i of places. Each place at stage $i+1$ takes as input the majority of the inputs that it receives from stage i. At this point the places in G_i have to know the places in G_{i-1} because otherwise malicious places could simply produce a majority of bogus agents and send them to the places in G_i. The agents thus have to carry a *privilege*. Bogus agents can then be identified and deleted. Two protocols have been proposed for implementing such privileges. Both require at each stage that a majority of places in G_i are non-faulty.

- *Shared secret.* These protocols ensure that only the source and the destination can learn a secret. Suppose we have a system where each stage has $2k-1$ places. Each stage i thus divides the secret into $2k-1$ fragments using a $(2k-1,k)$ threshold scheme. [5] Each fragment is then sent to a different place of G_{i+1}. To reconstruct the secret, a node needs at least k fragments of it.

- *Authentication chains.* In this scheme all agents carry unforgeable certificates describing their itineraries. Whereas only a place p can construct the certificate, if non-faulty, any place can check its validity. Every place uses sender authentication to reject bogus agents and selects any agent for which it received equivalent replicas from a majority of the places of the previous step.

This approach requires that multiple replicas of a server are available at each stage. In a real-world scenario it may be difficult to ensure this requirement.

The first three presented algorithms all focus on how to protect the results of an agent. Whereas Karjoth's algorithms allow to detect potential security attacks under the circumstances mentioned above, a TPE prevents tampering with the mobile agent once it is inside the TPE. In particular, it ensures the privacy mentioned in the second example. Sander's algorithm is currently too limited to be of general use, but allows an agent and a server to communicate in particular situations without disclosing confidential information.

The last algorithm prevents malicious places from inserting faulty results into an agent's execution. However, additional mechanisms, such as encryption, have to be integrated to prevent malicious hosts from accessing the agent's confidential data.

This discussion shows that the security problems of mobile agents are still not satisfactory solved and they are still subject to ongoing research.

5.5 Fault Tolerance

Information systems are subject to various potential failures. Besides hardware failures, processes may crash or implementations may be faulty. Applications have to cope with these problems and mask these failures from the user. The goal of fault tolerance is to transform an unfavorable and unpredictable behavior of the system into either a predictable behavior (e.g., atomic transactions) or to mask the unfavorable behavior (e.g., using replication). Both approaches will be discussed in this section. A simple classification of today's protocols is not always straightforward because they are often based on a combination of the two fault-tolerance techniques.

We distinguish between *site failures* and *communication failures*. Site failures can exhibit *crash-failure* and *byzantine* behavior. Contrary to crash-failure behavior, where the execution of a component is supposed to stop, a component with byzantine behavior behaves arbitrarily upon a failure. It potentially sends messages inconsistent with the protocol or executes a protocol only partly.
A node can recover after a failure. Nodes that have not crashed or have recovered are called *correct.*

[5] A (n, k) threshold scheme divides a secret into n fragments, where only possession of at least k fragments will allow the secret to be restored.

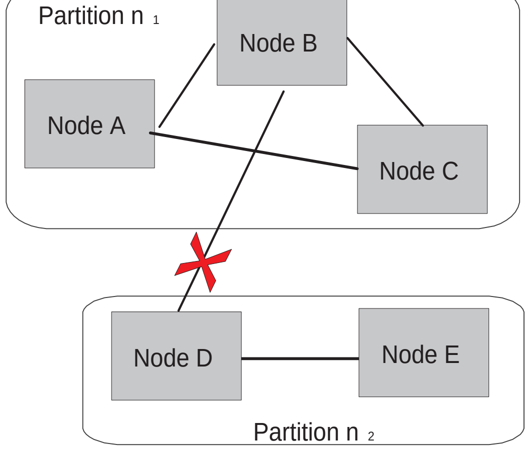

Figure 3: Communication failure between nodes B and C leads to the two network partitions n_1 and n_2.

Communication failures are more difficult to cope with. They may lead to a partitioning of the network (see Figure 3).

In an *asynchronous system* no prediction for the time it takes a message to go from a node n_1 to node n_2 is possible; no boundary on relative processor speed exists. For instance, if n_1 is waiting for n_2 to acknowledge a previously sent message, n_1 can not distinguish whether n_2 has failed or is merely slow. *Synchronous systems* rely on bounded transmission delays for messages as well as bounded relative speeds of processors.
A *reliable broadcast* of message m ensures that all correct processes eventually receive m.

It is very tempting to mask the problems of fault tolerance from the application developer. In the case of mobile agents this could mean that the fault-tolerance mechanisms are built into the places and therefore are invisible to the agent and the agent developer. However this incurs additional costs in terms of performance compared to the architecture where fault tolerance is explicitly built into the agent itself with the agent developer being responsible for the correct implementation.

We present the existing protocols classified according to the two approaches mentioned above, i.e., replication-based and transaction-based.

5.5.1 Replication-based Schemes

Schneider's protocol [Sch97], presented in Section 5.4.2, also supports fault tolerance even in the case of malicious nodes that appear to exhibit byzantine behavior from the point of view of the agent.

5.5.2 Transaction-based Schemes

Transaction-based schemes consider the agent execution as a series of transactions. The following protocols fall under this type. They only deal with crash-failure behavior.

Rothermel and Strasser [RS98] introduced a protocol that guarantees the *exactly-once property* of executions at each stage. They rely on transactional message queues [BN97]. When a mobile agent is launched, its actions are performed exactly once, even though its code could have been executed more than once in the case of failures. The execution at each state is as follows:

1. Get the agent from the message input queue.
2. Execute the code of the agent exactly once.
3. Put the agent into the message output queue to all the nodes of the next stage.

These actions together form a transaction. The execution of an agent consists of a sequence of transactions. Several nodes are responsible for the execution of a stage. One processes the agent (*worker*) while the others (*observers*) monitor its execution. If the worker node fails, the observers select another worker among themselves to perform the task, according to priorities assigned to the observers. As the observers, in general, can not distinguish between a communication failure leading to a partitioning of the network or the former worker having crashed, it may happen that the agent is executed twice. However, this would violate the exactly-once property. Therefore a stage transaction can only commit if a majority of the stage nodes agree. This is achieved by a special votation protocol built into the two-phase commit protocol (2PC) [BHG87]. The voting parties are called *voters*. They are located on every stage node (worker and observers). A so-called *orchestrator* organizes the vote. When asked to prepare for a commit (prepareCommit) by the local transaction manager (TM) [GR93] it requests the votes from the voters of the current stage. Upon receipt of a majority of YES votes the orchestrator returns YES to its local TM. The TM then decides that the transaction can commit and sends a commit to the orchestrator. After that, the orchestrator periodically sends a FORGET message to every voter and waits for *all* voters to acknowledge message receipt. At this point the protocol blocks if not all the nodes are correct. Upon receiving a FORGET message, the voters delete all evidence of the current stage's transaction locally. If the orchestrator gets an ABORT message from the TM it undoes the votation and restarts the transaction.
The protocol allows only for forward recovery.

[dASPZ98] improves this protocol by overcoming some of its limitations. In particular, it adds support for the simultaneous execution of more than one atomic transaction at a place, integrates distributed storage of recovery information, and allows partial recovery of an activity carried out at a place. A three-phase commit protocol [BHG87] replaces the 2PC protocol to reduce the probability of blocking.

Another approach has been suggested by Johansen et al. [JMS+98] based on detection and recovery. The agent's overall fault-tolerant itinerant computation consists of several *fault-tolerant actions* (FTAs), which are defined as

FTA: action A recovery $\overline{A}$

A is thereby called a *regular* action whereas $\overline{A}$ is a *recovery* action. The execution of FTA satisfies the following assumptions:

1. A executes at most once, either with or without failing.
2. If A fails, then $\overline{A}$ executes at least once and executes without failing exactly once.
3. $\overline{A}$ executes only if A fails.

A mobile agent's execution can be divided into a sequence of action with a_i being the i^{th} action. The single actions are separated by one of the following agent operations:

- *move*: The agent can move itself to another host.
- *spawn*: The agent copies itself.
- *checkpoint*: Allows the agent to ensure that its current state is stable.

The state of the mobile agent is called a *briefcase*. It travels with the agent. While in critical execution points, the agent is monitored by *rear guards*. Unlike the normal primary-backup approach they rather execute recovery code than the code executing when the program failed.
The NAP protocol is based on a reliable broadcast protocol. When execution stage a_i terminates, briefcase b is reliably broadcast to all rear guards of a_i, the place p_{i+1} responsible for executing a_{i+1} and its rear guards (see Figure 4). These destinations form the group G_i. The reliable broadcast thereby ensures that *all* correct places will either deliver b or not deliver b. If the rear guards of a_i

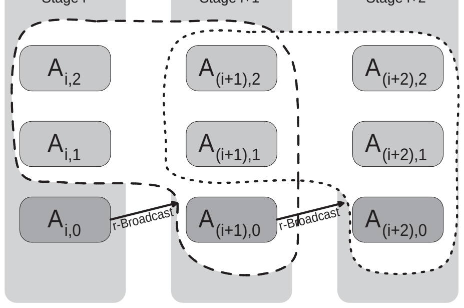

Figure 4: NAP algorithm.

receive b then they know that a_i has correctly terminated its execution and that the next stage of the agent execution has been triggered.
Even though this protocol does cope with site failures it does not handle communication failures leading to partitions in the network. In particular, the third assumption of the NAP protocol does no longer hold. If a partitioning occurs in a way that a rear agent can no longer communicate with its assigned place p_i, it will execute the recovery action $\overline{a_i}$ even though p_i might just be running in another partition.
Using an undo recovery action, a mobile agent's execution can be turned into a sequence of transactions.

5.6 Support for Global Transactions

In e-business but also in other applications it is mandatory to have transaction support for operations. For instance a mobile agent booking a flight with an airline company and renting a car at the destination wants to have both operations committed at the same time and not just one of them; if the flight cannot be booked it does not make sense to rent a car at the destination and vice versa.

This simple example clearly illustrates that both operations have to be executed as a transaction. An agent's execution is thus considered a global transaction, performed over the entire network. This global transaction consists of a set of subtransactions and is called an *agent-based transaction* [dASK97].

Owing to the heterogeneous and autonomous environment they operate in and their typically longevity, agent-based transactions have specific requirements [dASK97]. In particular, parts of the transaction should be allowed to commit before the global transaction commits. In addition, a failure during transaction processing should not automatically lead to an abort of the transaction. On the contrary, the agent should be able to recover from the failure and continue execution (e.g., on a replica of the current node). This recovery mechanism is based on the notions of *alternative tasks*, whose functionality substitutes a failed task, and *compensating tasks*, which cancel all effects of a committed task semantically.

With agent-based transactions, the execution of a subtransaction is subject to the ACID properties [GR93]. Isolated execution of a set of subtransactions can be achieved by ACID groups. Global consistency is achieved when the following properties hold:

- Places provide a *visible prepared-to-commit state*. This property enables the place to block the data until a commit or abort decision of the global transaction is received.
- The local scheduling is *rigorous*, i.e. no data item may be accessed until the previous transaction that modified it has been committed or aborted.

This approach greatly limits the overall transaction throughput in a system, as time-consuming transaction spanning multiple places may effectively block important parts of a system. Figure 5 illustrates a situation where both transactions T_l and T_k have to be aborted.

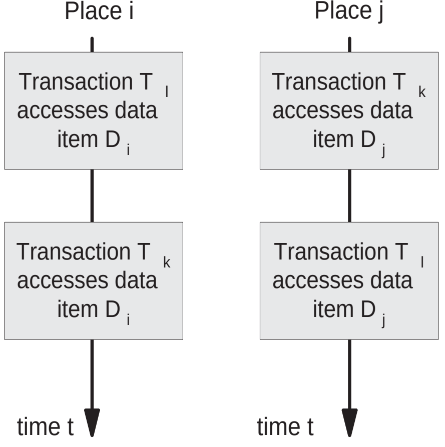
Place i
Place j
Transaction T_l accesses data item D_i
Transaction T_k accesses data item D_j
Transaction T_k accesses data item D_i
Transaction T_l accesses data item D_j
time t
time t

Figure 5: Non-serializable ordering of transaction execution.

As some applications do not require an equally high level of consistency, relaxing the consistency criterion enhances the overall performance of the system. Local consistency, for instance, merely ensures that the data at a place is always in a consistent state and in that case the transactions T_l and T_k in Figure 5 need not be aborted.

Another approach to guarantee the ACID properties for agent-based transactions uses a central coordinator that acts as the global transaction manager. It decides, based on some global correctness criteria, whether a global transaction can commit or whether it has to be aborted. However, this approach restricts the autonomy of mobile agents. In addition, it introduces a potential bottleneck into a system, making it difficult to scale.

Relying on distributed transaction management mechanisms such as transaction-processing (TP) monitors (e.g., Component Broker [IBMb], Tuxedo [ACDF96]) also requires the places to communicate with each other, thereby sacrificing autonomy and network bandwidth.

6 Standards

Because of their relatively recent nature, today's mobile agent systems differ widely in architecture and implementation. These differences prevent interoperability and rapid development of agent technology. To promote interoperability yet still permit system diversity, the Object Management Group (OMG)[6] is currently defining a standard for mobile agent technologies called the *Mobile Agent System Interoperability Framework* (MASIF or MAF) [Gro97, MBB+98]. This standard does not consider language interoperability but only interoperability between agent systems written in the same language by potentially different vendors. The following aspects have been standardized:

- *Agent management* contains actions such as agent creation given a class name for the agent, suspending, resuming, or terminating an agent's thread. Defining common management functions introduces a consistent management mechanism that applies to potentially many agent systems.
- *Agent transfer* allows an agent to travel to agent systems to facilitate a particular data exchange by exploiting locality.
- *Agent and agent system names* define the way an agent is identified.
- The *agent system types* define the profile of an agent. It may for instance include the manufacturer of the agent system, the agent language, and the serialization algorithm used.

[6]See OMG's homepage at http://www.omg.org

- *Location syntax* describes the syntax used to access agent system type information from destination agent systems.

The standards for mobile agent technologies rely on basic CORBA facilities such as naming service, life cycle service, etc. [OMG98b]. The standardization efforts for security issues, serialization formats, and execution state to enhance language interoperability have been delayed.

The complexity of the MASIF standard has been criticized as well as its focus on Java and its proximity to the IBM Aglets [IBMa] implementation. Tham et al. [TFWR97] claim that internal APIs have been too extensively specified and that the standard is weak in terms of interoperability and conformance specification.

Besides OMG, other organizations are also trying to standardize aspects of mobile agent technology:

- The Foundation for Intelligent Physical Agents (FIPA) [Fip] aims at maximizing the potential interoperability between different agent platforms. It addresses a variety of domains such as *agent management*, *agent security*, and *agent mobility*.

- The Agent Society's[7] goal is to define a common reference model for agent systems as well as to collaborate and coordinate work between different standard organizations.

7 Conclusions and Future Work

We have presented the current state of the art in mobile agent technology. Emphasis was placed on the fields of security, fault-tolerance, and transaction support. These fields are believed to be crucial for the deployment of this technology. They are still the subject of ongoing research activities, as up to now no complete satisfactory solutions have been proposed or implemented.

Today not many large-scale applications are based on mobile agents. This is partly because no "killer" application exists that would extend the use of mobile agents into many computation areas. However, in some domains its influence is growing. In active networks, for instance, packets carry code which is executed in the switches and allows the switches to be configured dynamically. Another application domain can be found in mobile computing. The agents act on behalf of mobile components and therefore minimize the connection time between the base station and the mobile device. Mobile agent technology is also believed to have a significant impact on e-commerce.

Our future work will focus on aspects of fault tolerance and transaction support for mobile agents, where we will address some of the open issues mentioned in Section 5. In this respect, the current paper helped establish the current state of the art in these fields.

References

[ACDF96] J.M. Andrade, M. Carges, T. Dwyer, and S. Felts. *The Tuxedo System: Software for Constructing and Managing Distributed Business Applications.* Addison-Wesley, Reading, Massachusetts, USA, September 1996.

[Bau97] J. Baumann. A protocol for orphan detection and termination in mobile agent systems. Technical Report 1997/09, Institut für parallele und verteilte Höchstleistungsrechner (IPVR), Fakultät Informatik, Universität Stuttgart, Breitwiesenstrasse 20-22, D-70565 Stuttgart, July 1997.

[BEA96] BEA System. *BEA Tuxedo: The Programming Model*, November 1996. White Paper, http://www.beasys.com/products/tuxedo/wps/index.htm.

[7]See Agent Society's homepage at http://www.agent.org

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, USA, 1987.

[BN97] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, San Mateo, CA, USA, 1997.

[CGH+95] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communication Systems*, 2(5):34–49, October 1995.

[CGPV97] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analizing mobile code languages. In *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222, pages 93–110. Springer Verlag, April 1997.

[CHK98] D. Chess, C.G. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In G. Vigna, editor, *Mobile Agents and Security*, LCNS 1419, pages 25–47. Springer Verlag, 1998.

[dASK97] F.M. de Assis Silva and S. Krause. A distributed transaction model based on mobile agents. In Kurt Rothermel and R. Popescu-Zeletin, editors, *Mobile Agents, Proceedings of the First International Workshop, MA'97*, LNCS 1219, pages 198–209. Springer Verlag, April 1997.

[dASPZ98] F.M. de Assis Silva and R. Popescu-Zeletin. An approach for providing mobile agent fault tolerance. In Kurt Rothermel and F. Hohl, editors, *Mobile Agents, Proceedings of the Second International Workshop, MA'98*, LNCS 1477, pages 14–15. Springer Verlag, September 1998.

[Fip] *Foundation for Intelligent Physical Agents (FIPA)*. http://www.fipa.org.

[Fip98a] Foundation for Intelligent Physical Agents (FIPA), Geneva, Switzerland. *FIPA 97 Specification, Version 2.0, Part 2: Agent Communication Language*, October 1998.

[Fip98b] Foundation for Intelligent Physical Agents (FIPA), Geneva, Switzerland. *Specification FIPA 98, Part 12: Ontology Service*, October 1998.

[FM96] J.S. Fritzinger and M. Mueller. *Java Security*. Sun Microsystems, Inc., 1996.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[Gen95] General Magic. *The Telescript Language Reference*, 1995. General Magic Inc., Sunnyvale CA.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[Gra96] R.S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Forth Annual Usenix Tcl/Tk Workshop*, 1996.

[Gro97] Object Management Group. Mobile agent system interoperability facilities specification. *OMG TC Document orbos/97-10-05*, November 1997. Update of Revised MAF Submission.

[IBMa] IBM. *Aglets Workbench*. http://www.trl.ibm.co.jp/aglets/index.html.

[IBMb] IBM. *Component Broker*. http://www.software.ibm.com/ad/cb/litp.html.

[JMS$^+$98] D. Johansen, K. Marzullo, F. B. Schneider, K. Jacobsen, and D. Zagorodnov. NAP: Practical fault-tolerance for itinerant computations. Technical Report TR98-1716, CS Dept., Cornell University, Ithaca, NY, USA, November 1998.

[KAG98] G. Karjoth, N. Asokan, and C. Gülcü. Protecting the computation results of free-roaming agents. In Kurt Rothermel and F. Hohl, editors, *Mobile Agents, Proceedings of the Second International Workshop, MA'98*, LNCS 1477, pages 195–207. Springer Verlag, September 1998.

[KZ97] J. Kiniry and D. Zimmermann. A hands-on look at java mobile agents. *IEEE Internet Computing*, July 1997.

[LF97] Y. Labrou and T. Finin. A proposal for a new KQML specification. Technical Report CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250, February 1997.

[LFP99] Y. Labrou, T. Finin, and Y. Peng. Agent communication languages: The current landscape. *IEEE Intelligent Systems*, pages 45–52, March/April 1999.

[LO99] D.B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communciations of the ACM*, 45(3):88–89, March 1999.

[Mat97] F. Mattern. Mobile agenten. Technische Universität Darmstadt, Paper in German, 1997.

[MBB$^+$98] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF, The OMG Mobile Agent System Interoperability Facility. In Kurt Rothermel and F. Hohl, editors, *Mobile Agents, Proceedings of the Second International Workshop, MA'98*, LNCS 1477, pages 14–15. Springer Verlag, September 1998.

[MGM99] P. Maes, R.H. Guttman, and A.G. Moukas. Agents that buy and sell. *Communication of the ACM*, 42(3):81–91, March 1999.

[NL96] G.C. Necula and P. Lee. Proof-carrying code. Technical Report CMU-CS-96-165, Carnegie Mellon University, Pittsburgh, USA, September 1996.

[Obj99] ObjectSpace. *Voyager: ORB 3.0 Developer Guide*, 1999. http://www.objectspace.com.

[OMG98a] OMG. *The Common Object Request Broker: Architecture and Specification, Version 2.2*, February 1998. http://www.omg.org/corba/corbaiiop.html.

[OMG98b] OMG. *CORBAservices: Common Object Services Specification*, December 1998. http://www.omg.org/library/csindx.html.

[PC97] C.P. Pfleeger and D.M. Cooper. Security and privacy: Promising advances. *IEEE Software*, September/October 1997.

[RG98] A.D. Rubin and D. E. Greer. Mobile code security. *IEEE Internet Computing*, November 1998.

[RS98] K. Rothermel and M. Strasser. A fault-tolerant protocol for providing the exactly-once property of mobile agents. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS), Purdue University, West Lafayette, Indiana, USA*, pages 100–108, October 1998.

[SBH97] M. Strasser, J. Baumann, and F. Hohl. Mole - a java based mobile agent system. In M. Mühlhäuser, editor, *Special Issues in Object Oriented Programming*, pages 301–308. dpunkt Verlag, 1997.

[Sch97] F.B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the 11th International Workshop on Distributed Algorithms, Saarbrücken, Germany*, September 1997. Invited paper.

[ST98] T. Sander and C.F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, LNCS 1419, pages 44–60. Springer-Verlag, 1998.

[TFWR97] C. Tham, B. Friedman, J. White, and T. Rutkowski. An assessment of the mobile agent facility proposal. http://www.agent.org/pub/satp/papers/maf-assessment.html, January 1997.

[TW96] D. Tennenhouse and D. Wetheral. Towards an active network architecture. In *ACM Computer Communications Review*, volume 26, pages 5–18, April 1996.

[Uni] University of Tromsø, Norway; Cornell University, Ithaca, New York, USA; University of California, San Diego, USA. *Tromsø and Cornell Mobile Agents (Tacoma).* http://www.tacoma.cs.uit.no/.

[WBS98] U.G. Wilhelm, L. Buttyàn, and S. Staamann. On the problem of trust in mobile agent systems. In *Symposium on Network and Distributed System Security.* Internet Society, March 1998.

[WPM99] D. Wong, N. Paciorek, and D. Moore. Java-based mobile agents. *Communication of the ACM*, 42(3):93–102, March 1999.