Security Issues in Agent Based Computing

Michel Abdalla[1]
Walfredo Cirne[2]
Leslie Franklin
Abdallah Tabbara

{mabdalla, walfredo, franklin, atabbara}@cs.ucsd.edu

University of California San Diego
Department of Computer Science and Engineering
9500 Gilman Drive
La Jolla, CA - 92093 - USA

Abstract

With the increase in popularity of computer networks, there has been a shift proposed in distributed system programming from the *remote procedure call* (RPC) to the *remote programming* (RP) paradigm, to decrease network traffic and improve performance. To accomplish this, several experimental systems use agents (i. e., mobile processes) to implement the RP paradigm. The use of agents has several advantages and a few disadvantages, including added security issues. In this paper, we survey several experimental agent systems under development and describe their current security mechanisms. We then develop a general agent model and discuss general security issues in that model. Finally, we propose new solutions that address some of these security issues.

Resumo

Com a crescente popularidade das redes de computadores, tem sido observada a tendência de substituir o tradicional paradigma de chamada a procedimentos remotos (RPC) pelo de programação remota (RP). O principal objetivo é diminuir o tráfego na rede e consequentemente melhorar o desempenho. Vários sistemas experimentais usam agentes (i. e., processos móveis) na implementação do paradigma de RP. O uso de agentes tem várias vantagens e apenas algumas desvantagens, estando entre elas os aspectos relacionados a segurança. Neste artigo, vários sistemas baseados em agentes atualmente em desenvolvimento são apresentados e seus respectivos sistemas de segurança, descritos. Um modelo geral para agentes é então desenvolvido e vários aspectos gerais de segurança relacionados a este modelo são discutidos. Finalmente, novas soluções que atendem a alguns destes aspectos são propostas.

1 Ph.D. student supported by CAPES (process BEX3019/95-2)

2 Professor of the Universidade Federal da Paraíba and currently Ph.D. student supported by CAPES (process BEX2428/95-4)

1. Introduction

The idea for agents comes from Artificial Intelligence (AI). Agents were developed as representatives of a user (thus the use of the word "agent"), which a program could "spawn" and delegate to perform a task independently of the program within a network. An agent can be thought of as a digital secretary with a "brain" to which one can assign tasks, then send it off to perform them. Because of this abstraction, they have a benefit of being intuitive for non-computer-literate people [Johansen 95a], due to the fact that people deal with agents all the time. For example, when many people plan a complicated vacation, they often do not contact the airlines, hotels, and car rental agencies directly. Instead, they interact with a travel agent who communicates with these companies on their behalf.

Beneath this abstraction, an agent is a process that can move from one machine to another by its own initiative. This is different from traditional process migration, where the transfer is initiated by the system. The ability for a process to move from machine to machine gives agents advantages over traditional processes, but it also leaves agent based systems open to additional problems.

Agents were created to simplify the parent process by allowing the computation to run on the machine where the resources needed to perform the task are located, instead of transferring these resources through the network. Generally, agents use the *remote programming* (RP) instead of the *remote procedure call* (RPC) paradigm to improve performance by eliminating intermediate network traffic [GenMagic 96]. There are some advantages to this arrangement. One advantage is that only selected results from complex calculations or queries will be sent back to the parent process, thus decreasing network traffic by not having to send commands, raw data, and intermediate results over the network as is done in some distributed systems.

A second advantage is that, since an agent knows how to access the data it needs to use on the host to accomplish its task, the host can be simplified in that it does not necessarily need to recognize a complex query language nor how to perform complex calculations. In other words, custom queries and calculations are the responsibilities of the agent and not the host, simplifying implementation of the host.

For example, one system under development that makes use of these two advantages is a database of 3-Dimensional (3D) Computer Assisted Drafting (CAD) files of mechanical parts [Cybenko 96]. In this system, when a draftsperson starts working on drawing a new part, first a rough outline is drawn in the CAD program. Then, agents are sent out to several database servers with a mathematical representation of the outline. There, the agents do complex calculations in an attempt to compare the rough drawing with the finished drawings in the database files on the remote servers. The agent at each server compiles a list of similar drawing file locations and relays that list back to the draftsperson's machine. After receiving replies from all of the remote agents, the program that initiated the agents displays a list of files, so that the user can select the drawing that is most similar to the one to be drawn. By using this system, the draftsperson only has to copy and modify a drawing to specification and does not have to redraw it from scratch. Furthermore, as complex calculations are being performed, data and intermediate results do not have to be shuttled back and forth across the network.

Another example is an experimental system that creates hypertext links to articles in a set of information resources [Cybenko 94]. The goal of this is to automatically create lists of links to related articles by using a latent semantic indexing (LSI) package to compare all articles with certain key phrases of a "pivot" article. Clusters of related articles can be found and articles that are similar enough to be "redundant" can be excluded. Instead of downloading all of the articles to a local machine to compare (since the servers are just information resources), agents are used similar to the CAD example above. In this way, the complex LSI computations are performed on the servers and just the resulting candidate articles are returned for inclusion into the cluster of links.

Another advantage of agents is that the machine where an agent originates does not have to be "online" in order for the agent to complete its task. This is especially useful in the case of mobile computing or in locations with poor or intermittent network access, where an agent's originating machine might be disconnected from the network for extended periods of time. In these cases, work can be done by the agent in the network during periods when the machine is disconnected.

For example, a system can be set up for mobile computers using "network-sensing tools and a docking system that allows an agent to transparently move between mobile computers, regardless of when the computers connect to the network" [Gray 96b]. In this system, if a computer is disconnected from the network (undocked) and an agent is sent to another machine to perform a task, the state of the agent can be stored and the agent can effectively be "put to sleep". When the computer is docked, the agent is "waken up" and enters the network to be routed to the machine where it can do its work. During this time, the mobile computer can be undocked and docked many times before the agent's work is done. When the agent completes its task, it can be routed back to the mobile computer. If the computer is not docked at the time the agent returns, it can "sleep" until the docking of the machine "wakes it". At that time, it can reenter the mobile computer to make the results available to the user. Note that these capabilities can also be used on a desktop computer that has an unreliable or intermittent network connection, such as via modem over an analog telephone line.

Agents can also be used by a client to make a network service available that did not exist previously. For example, if a client wanted to be able to perform queries on data that is located in multiple different database systems, an agent can be sent to one database system, perform a query, and send the results to another agent on another machine. That agent uses those results to make a query at a completely different database system. Those results can be relayed to another agent for further processing or to the user. In this way, a client can define custom services on a network. This could be done from the client's site without using agents, but there would be increased network traffic (and a corresponding decrease in performance) because all intermediate query results would have to be sent from the remote site.

As we have seen above, agents can also be viewed as another approach to building distributed applications, by providing another abstraction level to distributed programming in reducing the gap between distributed and centralized programming. They also potentially provide good support to mobile users, because they allow a simple model for disconnected execution.

Disadvantages to agents include extra required system services on the machines on which they can run and added security issues. In order to perform the tasks we have described, there are certain attributes that agents (and the systems they run on) must possess: 1. Machines on a network might have different architectures and will probably have different available resources. An agent must be flexible enough to run under these different types of environments. 2. Agents have the ability to move to another machine to continue execution, they must have some facility to save their code and state, send themselves to another machine, restore their code and state, and reinstate execution on the new machine. 3. Agents often have results to relay back to the parent program, they need methods to do this. These determine the extra system services that need to be provided by systems using agents. This will be described in greater detail in other sections of this paper.

With agents, there are added security issues. Agent systems need to worry about such things as potentially malicious agents in systems, potentially malicious systems under which agents are running, and potentially malicious machines agents pass through *en route* to their destinations.

In this paper, we survey several agent systems under development and their current security mechanisms. We then develop a general agent model and discuss general security issues in that model. Finally, we propose new solutions that address some of these security issues.

2. Current Agents Systems

One can build an agent-based application from scratch, using just the network services provided by the Operating System. However, it is much more convenient to use an *Agent Support Environment* (ASE) to provide the functionality that is common to many agent-based applications. Such systems supply services like communication, execution, and migration, allowing the developer to focus on the application details, leaving the infrastructure details to the ASE.

Although all ASE's have the same goal (namely: deliver high-level support for agent-based development), different systems provide dissimilar services and are based on different models of computation. In order to give the reader an idea of the kind of the services provided by the ASE's, this section briefly describes three systems that have been receiving attention: Tacoma [Johansen 95a] [Johansen 95b], Agent Tcl [Gray 95] [Gray 96a], and Telescript [GenMagic 96].

2.1. Tacoma

The Tacoma (Tromsø And COrnell Moving Agents) Project's main objectives are to investigate what services need to be provided in order to support easy building of agents (i. e., what exactly the ASE should be) and how agents can be used to solve problems traditionally addressed by operating systems. In the previous Tacoma versions, the agents were written only in Tcl. Tacoma currently supports Tcl, C, Scheme, Perl, and Python [Johansen 96], although Tcl is still the principal language.

The basic concepts of Tacoma are *agent*, *folder*, *briefcase*, *cabinet*, and *meet.* The agent is the computational (i. e., execution) unit. A folder is a list of elements, each of which is an uninterpreted sequence of bits. A briefcase is a collection of named folders. A cabinet is also a collection of named folders. Briefcases and cabinets differ in their mobility. Briefcases are mobile; agents can carry them when they migrate to another machine. A cabinet is statically stored in a given machine and does not move. Cabinets are used to permanently store information in a given machine. That way, an agent can leave state behind it when it moves around.

Meet allows an agent to run another agent, passing a briefcase as a parameter. The calling agent specifies where the invoked agent is to execute (the briefcase parameter contains a folder named HOST which specifies the target machine). Tacoma offers directory services through broker agents. These agents maintain a database on which services each agent can deal with. Other agents consult the broker in order to discover which agent provides a given service and where that agent is running. The Tacoma team is working on a scheduler to get rid of the manual HOST folder operations. This would make these details more transparent.

Note that the meet primitive is actually a remote execution feature. Agent migration is done by putting the agent code into the parameter briefcase and meeting an execution agent (usually, the Tcl interpreter) in the target machine. Tacoma does not provide automatic state migration at all. Agents need to capture the internal state by themselves. This approach is more flexible and easier to implement, although less transparent. More flexible because an agent can invoke a different agent, not just itself (i. e., not just move). Moreover, only the part of the state that is necessary for continuing the execution in the target machine has to be transported on the net. This is easier to implement and less transparent because Tacoma does not have to bother saving agent states, but agents do.

There is no communication mechanism other than meet (and this is more a remote execution mechanism than a communication one). However, meet has an option (called RPC option) that blocks the caller agent and allows the invoked agent to return a value to the former. The returned value is available in the RESULT folder, which is in the parameter briefcase used to meet the called agent. Unfortunately, Tacoma does not provide a way for an agent to exchange messages with those agents not invoked by it.

Tacoma can be viewed as the minimalist approach for ASE design, due to its lack of a communication mechanism and transparent state migration. However, it is not clear that the lack of these features makes agent-based development much harder. In fact, recall that one of the Tacoma's aims is to determine what support is needed to build agent-based applications.

2.2. Agent Tcl

Agent Tcl is an ASE that is under development at Dartmouth College. Its goal is "to address the weaknesses of existing transportable-agent systems" [Gray 95]. An Agent Tcl agent can currently only be written in Tcl, but Java support is being added into the system.

Agent Tcl was designed to be easily extended by adding new languages and network services. Its architecture has four levels (see figure 1). The agents themselves form the highest level. The lowest level consists of an API for each network service used to move agents and to provide inter-agent communication. Currently, only TCP/IP is supported.

The second level is a server that runs at each network site that maintains running agents. The server provides the migration facilities, communication primitives, and non-volatile store. All other services (such as scheduling, group communication, and directory) are provided by specialized agents.

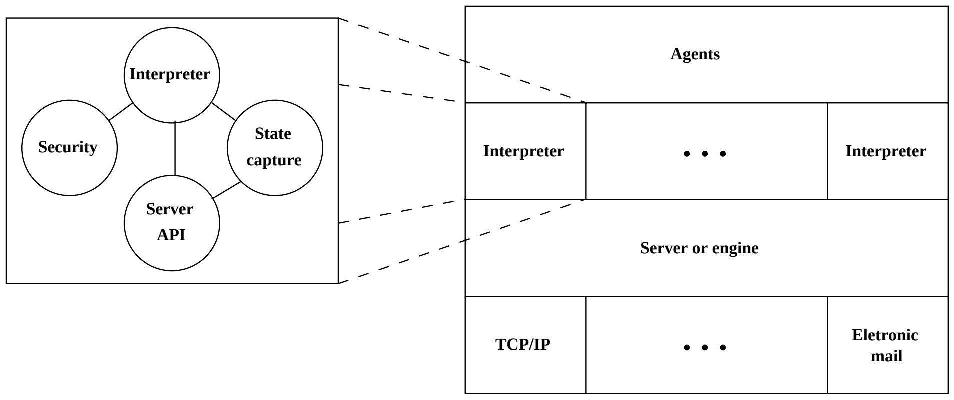

Figure 1. Agent Tcl Architecture (adapted from [Gray 95])

In Agent Tcl, the agent's state is implicitly transferred with the agent when it jumps from one machine to another. To make this transparent state transfer possible, Agent Tcl uses a modified version of the Tcl interpreter. The modification consists of separating the Tcl (i. e., the program being interpreted) stack from the regular C (i. e., the interpreter) stack so that the Tcl script and state can be saved and sent as a message to run on another machine. Unfortunately, the modified interpreter runs Tcl programs approximately 20 percent slower than the standard Tcl core.

An agent can send a message to another in two ways; through an event or via an established connection. An event provides asynchronous notification of an important occurrence while a connection is a named stream linking two agents. Events are not yet available.

The non-volatile storage permits agents to save their state, checkpointing their progress in order to make failure recovery easier. Note, however, that the failure recovery strategy is defined and implemented by the agents. The server only supplies the save/restore state commands.

The third architectural level consists of one execution interpreter for each available agent language. Each interpreter has four modules: the interpreter itself, a security supervisor that prevents an agent from performing forbidden actions, a state capture component that catches and reconstructs agents' internal states on the command of the server, and an API that interacts with the server to handle migration, communication, and checkpointing.

2.3. Telescript

Telescript is an object-oriented language that embeds facilities for building agent-based applications. Since it is a language-based solution, Telescript applications have to be written using only Telescript. No other language is supported. Telescript is commercially available by General Magic and is intended to be used as the basis for electronic marketplace networks.

A Telescript program consists of a collection of *classes*. Classes have *proprieties*, which are either *operations* or *attributes*, and can be either *private* or *public*. The public proprieties of a class describe its *interface*. Classes are hierarchically organized by sub-classing, and a limited form of multiple inheritance is available. *Objects* are instances of classes.

The three major concepts in the language are *agents*, *places*, and *go*. Both agents and places are processes. The difference between them is in their mobility: agents are mobile, can move about; places are stationary. Agents go to places, where they use places' services and/or interact with other agents that are in the same place. An agent always executes in the context of one or more enclosing place. Places provide a service API for agents to interact with. Places can be nested within other places.

Once in the same Telescript place, two Telescript agents can *meet*. A meeting lets agents (in the same place) call each other's operations. Meetings motivate agents to migrate. They permit all interactions between two agents to be done in a local (rather then remote) fashion. One agent initiates the meeting using the instruction meet. The agent specified in the meet accepts or declines the invitation.

While in different places, two agents can communicate by exchanging messages through a connection. One agent requests the connection, the other accepts or rejects it. The initiating agent identifies the responding agent, the place the latter occupies, and the quality of service required for the connection. Connections are a more traditional communication paradigm, but they can be useful for an agent-based application, especially when an agent needs some information from the user on who's behalf it is acting.

The Telescript language is interpreted rather than compiled. Telescript programs are interpreted by the Telescript engine, which provides all the functionality required by these programs, including the migration features. Therefore, the Telescript engine includes its own ASE. The Telescript engine is multitasked and isolates each agent or place in a separate Telescript process (not necessarily an OS process). Therefore, the purpose and progress of one agent have no influence, in general, over the purpose and progress of another.

Telescript models all resource accesses as requests to objects (i. e., a process can access a resource if it can send a request to the object which manages the resource). In addition to the objects defined in the language, other resources can be reached by augmenting the engine via the external applications API. This allows an external application (written, for example, in C) to interact with Telescript agents and places. An object can be transferred from one program to another, provided that the sender owns that object and all objects referenced by that object's properties, and recursively, all objects referenced by the objects in that object's closure.

3. Security Mechanisms in Current Agent Systems

Mobile Agents move from one machine to another and can execute on each of them. A major security problem in a network oriented environment is that neither the agent nor the machines are necessarily trustworthy. The agent might try to harm the machine and gain access to local resources. The machines might try to harm the agent or access its private information and resources. Either the machine or the agent may be malicious or badly programmed. However, this distinction is not of primary concern because the final effect can be the same. Security is perhaps the most critical issue in a mobile-agent system.

In the current agent systems, several different approaches are used to address these problems. There is a consensus that mechanisms should be provided to keep the machines from being harmed by the agents as well as to protect the agents from these machines. However, only a few systems implement some protection of the agents from the machines. In this section, the security mechanisms of the ASE's described in the previous section are presented.

3.1. Tacoma

In Tacoma, when an agent is visiting another site, it is treated as a guest program. In this system, all the agents visiting a foreign site must meet at an entry point at that site. The single entry point contains a firewall agent that can provide services like authentication, access control, accounting, and provision of fault-tolerance to the guest agent [Johansen 95b]. All agents arriving in a given machine are received by the firewall agent.

In the current implementation of Tacoma, the firewall agent basically logs the briefcase (which includes the agent code) to disk. This means the only security service provided is accounting. In order to be executed, the agent must be extracted from the briefcase. As the activation of an agent at the destination site may require a large amount of time, this task is not performed by the firewall agent. Instead, the firewall agent leaves this task to an execution agent. In this way, the firewall agent simply hands over the briefcase to the execution agent and is then ready for a new meet. The main purpose for this new level of indirection is to improve performance due to the fact that the firewall agent is not replicated, but the execution agent is [Johansen 95b].

3.2. Agent Tcl

In Agent Tcl, security issues are divided into four interrelated problems: to protect the machine from the agents, to protect agents from other agents, to protect the agent from the machine, and to protect a group of machines from the agent. However, in the current implementation, only the first two problems are addressed using authentication, authorization and enforcement. The remaining problems are left for a future implementation [Gray 96a].

During the authentication phase, Agent Tcl uses PGP (Pretty Good Privacy). PGP encrypts a message using the IDEA private-key algorithm and a randomly chosen private key, encrypts the private key using the RSA public-key algorithm and the recipient's public key, and then sends the encrypted key and file to the recipient.

During the registration of an agent with the server, a registration request is digitally signed using the owner's private key, encrypted using the server's public key, and sent to the server which in turn makes sure that the agent's owner is allowed to register on its machine. In order to prevent a malicious agent from masquerading as an existing agent during the session, the IDEA key is used for all further communication between the agent and the server. To ameliorate replay attacks, a sequence number is included in the messages.

During migration, an agent is digitally signed with the current server's private key and encrypted with the recipient server's public key. This migration message also includes the identity of the agent's owner. When accepted by a server, the apparent identity of the agent's owner, the authenticated identity of the sending server, and its degree of confidence in the agent's owners validity are recorded by this server. These same steps also occur when an agent communicates with another agent on a remote machine.

The current implementation has two major problems [Gray 96a]. The first is the absence of an automatic distribution mechanism for the PGP public keys. The second is that the system is vulnerable to replay attacks when an attacker replays a migrating agent or any exchanged message between agents.

After the authentication phase, an agent is submitted to an authorization phase, in which the system imposes access restrictions on the agent, and to an execution phase, in which these restrictions are verified and enforced.

Agents can have access to two different kinds of resources: indirect resources and built-in resources. In the former, the access can only be made through another agent. In the latter, access is granted directly through language primitives strictly for efficiency purposes.

For indirect resources, the relevant access restrictions are enforced by the agent which controls the resource. A quintuple is attached to each message from another agent. It contains the apparent identity of the agent's owner, the apparent identity of the sending server, a flag that indicates whether the sending server could be authenticated, a flag that indicates whether the owner could be authenticated, and its degree of confidence in the sending server.

For built-in resources, a generalization of Safe Tcl and a set of resource managers are used. With the use of Safe Tcl, two interpreters are provided for each incoming agent: a trusted and an untrusted interpreter. In the trusted interpreter, access to the standard Tcl/Tk commands are granted while only a smaller subset of them is accessible in the untrusted interpreter. All dangerous commands were removed from the untrusted interpreter and replaced by links to the trusted interpreter. A set of access lists, each containing the name of a resource manager along with the specified quantity of this resource, is maintained by the trusted interpreter.

In the current system, no direct access to the host resources is possible and there is no way for an agent to currupt the resource-manager system since there is no way for the agent to modify the access lists in the trusted interpreter. Therefore, the machine is well protected by Agent Tcl using the simple kernel-user model of Safe Tcl [Ousterhout 96].

3.3. Telescript

Security in Telescript is also a major concern. In this system, the server wants to be protected from an incoming malicious agent. On the other hand, the agent wants to have its information protected while it is traveling from one machine to another. Each place in the system might have its own policies while each engine has an overall policy.

Two concepts are fundamental to the understanding of this system: safety and security. The term safety refers to features which mainly promote robustness and prevent accidents. The term security, on the other hand, refers to features which are intended to provide protection and integrity in the presence of malicious users. These security features protect agents and places from each other.

Every agent is uniquely identified by a telename. A telename consists of two components: an authority and an identity. The authority identifies the owner of the agent. The identity distinguishes an agent from another agent of the same authority. The authority component is cryptographically generated and cannot be forged [Valente 94].

Telescript processes cannot directly access the resources of the machine they are running on. The instructions from which a program is constructed are not those of any “real” computer, nor are the objects the program manipulates those of any “real” operating system. Consequently, places and agents lack even the vocabulary required to directly examine or modify the memory, file system, or other physical resources of the computers on which they execute.

Access control in Telescript is based on the use of capabilities and is enforced by the use of permits. Each agent has a permit which limits its capability and the resource consumption. In this way, agents and places can be protected from malicious or badly programmed agents. Two kinds of capabilities are granted an agent by its permit. The first kind is the right to execute certain commands. The second is the right to use a particular resource and by which amount. An agent’s permit is granted when the agent is first created and is renegotiated whenever that agent migrates to another place with a different administrative authority.

Besides access control, secure channels are provided to support agent mobility in a distributed process environment. These channels provide an authenticated “opaque pipe,” normally created using cryptography [GenMagic 95]. Depending on the specific application, different levels of security can be provided. If authentication is required, strong mutual authentication using RSA public key encryption, session key negotiation, and session encryption is used [GenMagic 95].

4. A Model for Agent Based Computing

After describing the current implementation of agent systems and discussing the way these systems tackle certain security problems. In this section, we will define a generalized ASE model by providing many of its features. This model will then be used to explore the main security problems in agent based systems based on a classification presented in [Gray 96a].

4.1. The Model

An agent model that can be abstracted from the main descriptions in the previous sections is based on being able to build an ASE that is complete. This support system, as one can notice from the different implementations that were presented, needs to support the following main features, which define what an agent is by defining what its capabilities are.

4.1.1. Creation

This involves the ability of certain agents in creating other agents, to perform certain specified tasks. The resulting agents can be created to run localy or remotely.

4.1.2. Execution

Agents need to be able to execute in order to carry out their tasks. Execution is usually done through an interpreter that supports a runtime environment within which an agent can function. This environment is usually specified when the agent starts up in the machine.

4.1.3. Resource Access

This should implement ways in which an agent accesses certain local resources as well as resources being carried with the agent. What this implies is that there are two types of resources present within the execution environment, resulting in two very different security concerns. The types of resources being considered could include physical resources as well as logical resources and controlling access to those resources can be done using many different mechanisms including access control or capabilities. An example would be restricting access to the CPU and other resources through checking during the interpretation/execution phase.

4.1.4. Migration

This implements how an agent can move around from one machine to the next on its own initiative, while carrying out its task. This aspect of the architecture is what makes agents very interesting and what gives them unique properties. This migration process can be affected by the move being unrestricted (i.e. from any one machine to another) or restricted (i.e. from the home machine to the server machine and back). It also involves how and in what form the data is being carried around within the agent. This point refers to migrating agent resources (in addition to code migration). There is also the issue of implicit versus explicit transfer of an agent's state.

4.1.5. Communication

This takes care of the fact that agents need to interact with other agents to provide their services. There are two types of communication that could be available. One type would be local while the other is remote. Communication may also serve as a way to transfer objects (and agents) between other agents. This also includes the issue of creating synchronization primitives between agents that can help in organizing their efforts when obtaining a certain service in a cooperative manner.

4.1.6. Language Support

This involves the issue of interpreted versus compiled languages. It also involves support for just one specialized language versus the support of many different languages to be used in programming agents.

4.1.7. Additional Services

Such services like authentication, name service, check-pointing, as well as other system built-ins. The issue here is that some of these services can be implemented through the system while others are easier to implement using agents.

Note that all of the above mentioned features can help define an agent architecture and thus provide a framework within which all current research can be viewed and security issues can be discussed. The following is a description of the security issues associated with these different features of an agent based system.

4.2. Security in Agent Based Computing

It has been previously stated that security is the most important issue on which the applicability of agent systems rides [Chess 95] [Gray 96a] [Ousterhout 96] [Ramusson 96a] [Ramusson 96b]. No matter what the features of an implementation are, if they can't provide an adequate security model for all the issues involved, then this implementation will definitely fail. There has even been a suggestion towards the need for "security profiles" in agent systems to see if any given system adequately addresses all the issues concerned [Broune 95].

In current operating systems, the system resources are the most important parts of the system and thus should be protected from malicious use. It is acceptable that, in such a system, the solution for the security problems is based on splitting the domain into a user space and a kernel space. This is not the case in an agent system where the user has a vested interest in the agent and part of its resources which the system should not have control over.

In order to put these ideas into perspective, we will describe the main points for a good security system for agents. As stated above in a description of a model of the agent architecture, the system is made of a group of machines which provide a running environment for a group of agents. The points described address the problems of protecting these different parts of the system from one another [Gray 96a].

4.2.1. Protecting the Machine from Agents

This point is very similar to current operating systems concerns and the solutions suggested in the different research projects present the idea that operating systems solution are applicable in this case. One has to note here that static checking of a program has been proven to be an unsolvable problem and thus other approaches have to be used [Chess 95]. This problem seems to be basically a problem of controlling access to the different resources on the machine where the agent is running.

In certain specialized cases, access to certain interpreted commands has to be restricted and should thus be treated in a similar fashion to system calls. Resources that should be managed include communication, disk access, CPU time, screen access, etc. These have been grouped in [Ousterhout 96] into two groups:

- Resource access: where modifications are being made to the system in unauthorized ways. It is suggested that in order to prevent this kind of attack, certain operations have to be checked by adding another level of indirection in accessing the resource. This is one of the main reasons why interpreted languages became a very attractive vehicle for implementing agents. The interpreter can check the execution dynamically.
- Information access: where information is being stolen or leaked without the knowledge of the system since no malicious behavior affecting the machine is being performed. One way to solve this problem is to separate access to restricted data from access to the outside world, but this restricts agents in a very big way.

It is has been suggested that in order to control access effectively the identity of the agent should be determined, because this will allow the application of different access policies to different agents. This presents the problem of authentication, which in this case is a mapping between the agent and its owner. Note that the owner could be a non-registered user on that machine and could still be allowed certain access. But, authentication in a distributed system is a very difficult problem, especially if the solution needs to be scalable, which is the case here. On a smaller scale, authentication can work a little better and maybe it can be applied at different levels to support bigger systems.

Another problem is setting access permissions based on these authentications. This could be implemented using capabilities which seem to scale much better than access control lists in big systems, based on the argument of distributed versus centralized control. Currently proposed digital cash protocols can be used to provide an even better functionality for limiting capabilities from being passed along, based on n-spent coin schemes [Ferguson 93]. On the other hand, access control lists can also be used in a limited fashion in a small system.

4.2.2. Protecting Agents from Agents

This point is also very similar to its counterpart in operating systems, where different protection domains are specified to limit the direct access of one process to the internal state of another. The protection scheme may depend on checks done during domain changes. Because of the associated risks, if we assume that this direct access is not allowed in agent systems, then the main route for such an access between agents is through messages.

Thus, inter-agent communication has to be made secure for the transfer of objects and other agents. This problem is also addressed in many current agent systems where the solution resembles that of operating systems, by specifying restricted regions of access for each agent and communicating via messages. Note, that in the case of agents, there is also the issue of stealing resources or information from other agents, which makes the problem more general due to the presence of resources within the agents.

Also, there are two types of communication that can be supported (local and remote), and this will create different ways in which agents can interact and affect one another, unless the messaging interface is the same which is usually not the case. Authentication can help with establishing such communication schemes that are reliable. Another point is that communication in these systems generally goes beyond information exchange into a transactional model and thus is a more critical issue than just limiting access to the different agents. Note that such issues are not properly addressed in some current systems.

4.2.3. Protecting Agents from Machines

This is one of the main points that has been overlooked in current operating systems. On the other hand, the necessity of this feature has been argued in some of the current work that deals with theoretical issues in agent systems [Chess 95] [Gray 96a]. We believe that this point will determine the effectiveness of future agent system implementations. It is the Achilles' heel of agent based architectures. This is because most of the users are represented by agents, and thus have a vested interest in their agents' proper functioning and in the safety of the resources that are being carried around in them (like digital cash or information). For example, if a user wants to send out an agent in charge of buying some goods, that user has to provide the agent with some (digital) money or credit. A good buyer agent should shop around to find the best price before actually purchasing the product. During this tour, it can be robbed by a malicious host if necessary care to prevent this situation is not taken. Thus, even if all the other problems of agents architectures are solved satisfactorily, this problem would still be enough to make agent architectures fail in practical use.

The main point here is that the machine should not be able to tamper with an agent running on it, by changing its functionality or pulling sensitive information from it. But, since the agent depends for its existence on the machine it is running on, such a security problem could be hard to solve. Unless there is a piece of trusted hardware (i.e. the agent has its own processor to run on within that machine), this problem cannot be solved.

One suggested scheme detects tampering as the agent leaves the untrusted machine to a trusted one [Gray 96a]. Another would be to try and encrypt certain contents of an agent where the key to decrypt cannot be contained in the agent itself (or else the machine would figure out how to decrypt it) [Chess 95]. In any case, this problem is really hard to solve and thus far has been ignored by all current agent system implementations, increasing the importance of tackling that problem. Another suggested scheme is to authenticate the machine before moving to it, while another is to use an audit trail to figure out if problems had occurred after the fact [Gray 96a] [Johansen 95a], but all of these solutions are not enough to resolve a problem when it occurs.

Note that the issue here also includes privacy of the data that was collected by the agent. This is the reverse of the usual security problem, where the logical resources are not being used to abstract physical resources on the machine, but instead belong to the agent and the machine cannot be trusted with them.

4.2.4. Protect a Group of Machines from an Agent

The idea here is to avoid the case where a group of agents uses up a lot of available resources. This can be stated in another way by making the point that an agent can have control over an excessive number of resources as a whole even though it only has control over a few resources locally on each of the machines. This excessive control over resources could put those machines in a position where they are forced to behave in a certain way due to external pressures (e. g., slowing down a group of machine by overloading them). Another problem is that information could be deduced or compromised by accessing many sites and having a more global view by combining much complementary information. A good security policy should be able to alleviate such problems by restricting access on many different machines, and this can’t depend on local mechanisms but on global ones. One such method could use digital cash methods to limit access to these resources [Gray 96a].

Note that, as stated before, only the first two categories of problems have been addressed in current implementations, although all four have been recognized as valid concerns. This could possibly be due to how hard these problems are. The next section will present a practical solution for some of the agent based computing security problems discussed here.

5. A Practical Solution

Most of the research about security on Agent Based Computing focus on the “protect the machine from agents” and “protect an agent from other agents” problems [Sun 95] [Ousterhout 96] [Johansen 95b]. Moreover, those works that address the other secure agent-based computing issues either intend to solve the problem but don’t have a real solution yet [Gray 96a] or just mention that these problems are important and need to be solved [Chess 95].

Here, we propose a solution that also addresses the “protect the agent from the machine” issue, which is a very important concern from the user’s point-of-view. One major design decision is to conceive a practical solution, implementable using current technology. Therefore, our solution is both simple to understand and to implement, and does not depend upon a net-wide service, like a secure public key distribution system. The PEM acceptance problem (Privacy-Enhanced Mail [Linn 93]) has shown that this kind of new distributed net-wide directory infrastructure turns into a barrier to the adoption of the solutions based on it [Braden 94]. Note that if such an infrastructure became available, the rationale for this choice changes completely. But we don't expect this to be the case in the next few years.

Another important design decision is that an agent not concerned about security should not incur any security overhead. In other words, our goal is to design a flexible solution that does not require the use of the security mechanisms by the agents which don't need them. Thus, all the schemes described here are to be added to (without substituting) the traditional features found in an ASE (defined the section 4).

Our approach to the "protect the machine from agents" problem is the common one: the agent runs in a restricted environment and all accesses to resources have to pass through a security monitor. This monitor decides if the access is to be allowed or not. Note that this is not difficult to implement, especially if the agent is interpreted. For example, the Agent Tcl architecture (see section 2.2) provides a framework to implement this kind of monitor.

The problem resides in deciding whether a particular agent has access to a given resource. An easy solution is defining the same rights to all agents (as Java does [Sun 95]). However, this is a very restrictive solution. A better approach would be to grant different accesses depending on who owns the agent. Since we have decided not to use a distributed authentication service, we have to rely on passwords to securely identify the agent owner. Protocols that safely negotiate cryptographic keys (like SSL [Freier 96]) must be used to avoid the discovery of passwords by monitoring the network. Default access rights can be supplied for unknown users. Of course, a group of hosts can share the same password database, but our solution doesn't require this nor does it define how this can be done. We call this step the *authentication phase*.

If every host (or group of hosts) has to know the pair (user, password) for each user that has special access to it, there is no scale advantage in using capabilities. Thus, we choose a conventional Access Control List (ACL) mechanism to decide if an agent can or cannot access a resource, given that the system knows who owns the agent.

It is clearly impossible to protect the agent from the machine on which it runs. An agent executes when and if its host machine wants. Furthermore, the machine can change the agent in any way it wants. An agent needs to care about this question <u>before</u> it moves into a machine. Our solution requires that the machine sends a counter-password to the agent during the authentication phase. This means that both agent and target machine need to know a triple (user, password, counter-password) and allows the agent to determine if it is moving to a known machine.

In spite of that, an agent can still have some critical information stolen by a known machine. The shopping around agent is a perfect example of that situation. One of the digital stores the agent visits can steal the money being carried by the agent. To avoid that, we require that an agent concerned about security return back to its home (and so, completely secure) site after it finishes its job at the target site. Additionally, an agent is to carry only the resources and information it may need into the destination host. Observe that this implies that a secure agent cannot move about; it has to do a single round-trip. However, the same kind of service can still be provided because an agent can travel again after its returns to its home site. Actually, it has to be modified to carry useful information in the new destination, but this also doesn't limit the development of any type of agent based application. We call such a specialized, restricted agent a *minimal agent*.

The "protect an agent from other agents" problems can be seen as a special case of the "protect the machine from the agents" problem, except for the communication issue. One could think of using the same password plus ACL solution to cope with the secure communication issue. But, because any message could be intercepted by the host which is running the agent (even if the message is encrypted, the host can wait until the agent decrypts it), this is not worthwhile. Thus, we don't provide a special mechanism to make message-based communication more secure. If an agent needs this kind of service, it should migrate to the destination host (returning first to the home host) and communicate locally.

"Protect a group of machines from an agent" is a very difficult problem because even if the resource hoarding agent is detected and killed, the owner of that agent can send a new instance of it again. Consequently, it is necessary to determine who the owner (or at least, the originator machine) of the resource gathering agent is and log this information in order to figure out who is creating this kind of malicious agent. The problem is how to safely and scalably know who the owner of a particular agent is or from which machine it has been sent. Because, in our solution, we can determine this just for the agents that are using the services proposed here, there is no way to solve this problem for the general case of completely mobile agents.

The currency-based resource-allocation scheme suggested by Gray (see section 5) doesn't solve this problem in all cases either. In many cases, the machines on which an agent can run don't want to ask for some form of electronic currency because it is in the machine's best interest that the agent run and obtain as much information as it wants. The perfect example is a digital shopping center. In these cases, currency schemes don't really work.

6. Conclusions

Agents seem to be a very useful approach in building a large set of network applications. However, the security problems that can be raised by supporting agents can prevent the wide use of agent based applications. Moreover, distributed security concerns tend to become more important as we begin to use open computer networks to transfer information of more direct economic value. Therefore, a secure way to use agents is fundamental to make viable their application in public networks, like the Internet.

We presented here some of the motivations for agent based computing, as well as a brief introduction about the systems that have been receiving more attention by the research community and describe their security characteristics. This supplied the basis to construct a general model for the Agents Support Environment and to deeply discuss what the security requirements to these systems are.

We also proposed solutions for some of the problems identified in the analysis of the security requirements of agent based computing. The novel element in our solution is that we do not try to build a net-wide secure system; we rather use a truly distributed approach, in which each host is responsible for authenticating the agents that have special privileges on it and, symmetrically, each agent authenticates the destination machines when it carries some valuable information. Note that this solution avoids using global authentication services, and thus can be implemented in today's networks.

However, the authentication does not guarantee that the agent will not be attacked by the machine. In order to reduce the damage that such kind of attack could incur, we introduce the concept of minimal agent, which carries only the information it may need into the host and always returns back to its safe home host when its job is done.

It is clear that our solution involves some restrictions in the way agents can be used. Nevertheless, any application that can be built in the standard “insecure” agent model, can still be built using our more restricted services.

The agents security problems are very hard. An appealing alternative way to address these problems is the soft security approach [Rasmusson 96a] [Rasmusson 96b]. Soft security means that privileges are granted as they are needed, with the current risks taken into consideration. As opposed to soft security, hard (i. e., traditional) security uses methods that don’t reevaluate granted privileges.

A soft security mechanism, instead of using only the agent’s access rights, can grant or deny resources depending of the agent behavior. In this way, well-behaved programs would be allowed to do their jobs, while malicious programs would be detected and stopped. This idea was derived from works on intrusion detection in computer systems, which is concerned with finding activities that indicate that someone is using the system in a different/illegal way. An important characteristic in soft security systems is that they should present graceful degradation, which means that even if parts of the system are tricked by an agent, other parts should signal in the event of suspicious activities. Therefore, there should not be any way to obtain unrestricted access by attacking well known weaknesses of the system. Another nice feature of this approach is that it does not require any kind of centralized authentication service. It seems that soft security techniques can be especially useful to augment the default access rights in a hard security system, as the one we preposed.

References

[Braden 94]	R. Braden, D. Clark, S. Crocker, and C. Huitema. *Report of IAB Workshop on Security in the Internet Architecture*. Internet RFC 1636, 1994. ftp://ftp.internic.net/rfc/rfc1636.txt.
[Browne 95]	Shirley Browne. *Need for a Security Profile for Agent Execution Environments*. CIKM’95 Workshop on Intelligent Information Agents, 1995. http://www.cs.umbc.edu/%7ecikm/iia/submitted/viewing/browne.html.
[Cai 96]	Ting Cai, Peter A. Gloor, Saurab Nog. *DartFlow: A Workflow Management System on the Web using Transportable Agent*. 1996. ftp://ftp.cs.dartmouth.edu/TR/TR96-283.ps.Z.
[Chess 95]	David Chess, Benjamin Grosof, Colin Harrison, David Levine, and Colin Parris. *Itinerant Agents for Mobile Computing*. Research Report RC 20010. IBM Research Division, 1995.
[Cybenko 94]	George Cybenko et al. *Information Agents as Organizers*. Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94), 1994.
[Cybenko 96]	George Cybenko, Aditya Bhasin, and Kurt D. Cohen. *Pattern Recognition of 3D CAD Objects: Towards an Electronic Yellow Pages of Mechanical Parts*. 1996. http://comp-engg-www.dartmouth.edu/~3d.
[Ferguson 93]	Niels Ferguson. *Extensions of Single-term Coins*. Crypto’93, pp. 292-301, 1996.
[Freier 96]	Alan O. Freier, Philip Karlton, and Paul C. Kocher. *The SSL Protocol: Version 3.0*. Internet Draft, 1996. http://home.netscape. com/eng/ssl3/ssl-toc.html.

[Gray 95] Robert S. Gray. *Agent Tcl: A Transportable Agent System*. In Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95), December 1995. http://www.cs.dartmouth.edu/~agent/papers/cikm95.ps.Z.

[Gray 96a] Robert S. Gray. *Agent Tcl: A Flexible and Secure Mobile-agent System*. In Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96), July 1996. http://www.cs.dartmouth.edu/~agents/papers/tcl96.ps.Z.

[Gray 96b] Robert S. Gray et al. *Mobile Agents for Mobile Computing*. 1996. ftp://ftp.cs.dartmouth.edu/TR/TR96-285.ps.Z.

[GenMagic 95] General Magic. *An Introduction to Safety and Security in Telescript*. 1995. http://www.genmagic.com/Telescript/security.html.

[GenMagic 96] General Magic. *Telescript Technology: Mobile Agents*. 1996. http://www.genmagic.com/Telescript/Whitepapers/wp4/whitepaper-4.html.

[Johansen 95a] Dag Johansen, Robert van Renesse, and Fred Scheidner. *Operating system support for mobile agents*. In Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems, 1995. http://cs-tr.cs.cornell.edu/TR/CORNELLCS:TR94-1468.

[Johansen 95b] Dag Johansen, Robert van Renesse, and Fred Scheidner. *An Introduction to the TACOMA Distributed System: Version 1.0*. Technical Report 95-23, Department of Computer Science, University of Tromsø. 1995. http://www.cs.uit.no/Lokalt/Rapporter/Reports/9523.html.

[Johansen 96] Dag Johansen, Robert van Renesse, and Fred Scheidner. *Supporting Broad Internet Access to TACOMA*. In Proceedings of the Seventh ACM SIGOPS European Workshop, Connemara, Ireland, pp. 55-58. 9-11 September 1996. http://www.cs.uit.no/DOS/Tacoma/tacoma.webpages/SIGOPS.tac-www.ps.

[Linn 93] J. Linn, S. Kent, D. Balenson, and B.Kaliski. *Privacy Enhancement for Internet Electronic Mail: Parts I-IV*. Internet RFC 1421-1424, 1993. ftp://ftp.internic.net/rfc/rfc142[1-4].txt

[Ousterhout 96] John Ousterhout, Jacob Levy, and Brent Welch. *The Safe-Tcl Security Model*. Sun Microsystems Laboratories, 1996.

[Rasmusson 96a] Andreas Rasmusson and Sverker Jansson. *Personal Security Assistance for Secure Internet Commerce*. New Security Paradigms'96 Workshop, 1996. http://www.sics.se/~ara/doc/NSP/NSP.html

[Rasmusson 96b] Lars Rasmusson and Sverker Jansson. *Simulated Social Control for Secure Internet Commerce.* New Security Paradigms'96 Workshop, 1996. http://www.sics.se/~lra/nsp96/nsp96.html.

[Sun 95] Sun Microsystems. *The Java Language: An Overview*. 1995. ftp://ftp.javasoft.com/docs/java-overview.ps.

[Valente 94] Luis Valente. *Safety in Telescript*. 1994. http://catless.ncl.ac.uk/Risks/15.39.html#subj6.