D'Agents: Security in a multiple-language, mobile-agent system

Robert S. Gray[1] and David Kotz[2] and George Cybenko[1] and Daniela Rus[2]

[1] Thayer School of Engineering, Dartmouth College, Hanover NH 03755, USA
[2] Department of Computer Science, Dartmouth College, Hanover NH 03755, USA

Abstract. Mobile-agent systems must address three security issues: protecting an individual machine, protecting a group of machines, and protecting an agent. In this chapter, we discuss these three issues in the context of D'Agents, a mobile-agent system whose agents can be written in Tcl, Java and Scheme. (D'Agents was formerly known as Agent Tcl.) First we discuss mechanisms existing in D'Agents for protecting an individual machine: (1) cryptographic authentication of the agent's owner, (2) resource managers that make policy decisions based on the owner's identity, and (3) secure execution environments for each language that enforce the decisions of the resource managers. Then we discuss our planned market-based approach for protecting machine groups. Finally we consider several (partial) solutions for protecting an agent from a malicious machine.

1 Introduction

A mobile agent is a program that moves from machine to machine and executes on each. Neither the agent nor the machines are necessarily trustworthy. The agent might try to access or destroy privileged information or consume more than its share of some resource. The machines might try to pull sensitive information out of the agent or change the behavior of the agent by removing, modifying or adding to its data and code. A mobile-agent system that does not detect and prevent such malicious actions can never be used in real applications. In an open network environment, intentional attacks on both machines and agents will start as soon as the system is deployed, and even in a closed network environment with trusted users, there is still the danger of misprogrammed agents, which can do significant damage accidentally. Security is perhaps the most critical issue in a mobile-agent system. We consider the following four inter-related problems:

- *Protect the machine.* The machine should be able to authenticate the agent's owner, assign resource limits based on this authentication, and prevent any violation of the resource limits. To prevent both the theft or damage of sensitive information and denial-of-service attacks, the resource limits must include access rights (reading a certain file), maximum consumptions (total CPU time), and maximum consumptions per unit time (total CPU time per unit time).

A chapter in the book "Mobile Agents and Security", pages 154-187, edited by Giovanni Vigna.
Available at <http://agent.cs.dartmouth.edu/papers/gray:security-book.ps.Z>.

- *Protect other agents.* An agent should not be able to interfere with another agent or steal that agent's resources. This problem can be viewed as a subproblem of protecting the machine, since as long as an agent cannot subvert the agent communication mechanisms and cannot consume or hold excessive system resources, it will be unable to affect another agent unless that agent chooses to communicate with it.
- *Protect the agent.* A machine should not be able to tamper with an agent or pull sensitive information out of the agent without the agent's cooperation. Unfortunately, without hardware support, it is impossible to prevent a machine from doing whatever it wants with an agent that is currently executing on that machine. Instead we must try to detect tampering as soon as the agent migrates from a malicious machine back onto an honest machine, and then terminate or fix the agent if tampering has occurred. In addition, we must ensure that (1) sensitive information never passes through an untrusted machine in an unencrypted form, (2) the information is meaningless without cooperation from a trusted site, or (3) that theft of the information is not catastrophic and can be detected via an audit trail.
- *Protect a group of machines.* An agent might consume excessive resources in the network as a whole even if it consumes few resources at each machine. Obvious examples are an agent that roams through the network forever or an agent that creates two child agents on different machines, each of which creates two child agents in turn, and so on. An agent and its children should eventually be unable to obtain any resources anywhere and be terminated. If the network machines are under single administrative control, solutions are relatively straightforward; if the machines are not, solutions are much more complex.

Outline of this paper. Over the past few years we have developed a multi-language mobile-agents system, D'Agents, formerly known as Agent Tcl. A significant component of that effort has been to design and implement security mechanisms and policies to deal the issues described in this section. In this chapter we cover each of the above issues in turn, first describing how D'Agents addresses the first two concerns, and then briefly discussing some possible solutions for the second two concerns. Throughout our discussion we make a careful effort to distinguish between the architectural features of the D'Agents system, particularly its security *mechanisms*, and the security *policies* that are or could be implemented within that framework.

2 Related work

Although all of the problems discussed above have been considered in the mobile-agent literature [18, 4, 28, 22], most mobile-agent systems address only the first two problems, namely, protecting a machine from malicious agents and agents from each other. A growing number of mobile-agent projects, however, are experimenting with techniques for protecting machine groups from malicious agents and protecting agents from malicious machines. Here we consider some representative mobile-agent systems.

Telescript [31, 33, 32, 34], later marketed as part of the Tabriz web-server package and then withdrawn from the market, was the first commercial mobile-agent system. It has two security mechanisms. First, each agent carries cryptographic credentials so that the system could verify the identity of the agent's owner. Second, each agent carries a set of permits that give it the right to use certain Telescript instructions and certain amounts of available resources. Each machine imposes a set of permits on incoming agents to prevent that agent from taking undesired action. Agents that attempt to violate their permits are terminated immediately. Some permits involve resources that are "distributed" across multiple machines (e.g., maximum number of times that an agent can migrate). Since Telescript assumes that all machines are trustworthy, these permits are simply counters that are decremented as the agent travels through the network.

Nearly all mobile-agent systems protect the machine in the same manner as Telescript: (1) cryptographically verify the identity of the agent's owner, (2) assign access restrictions to the agent based on the owner's identity, and (3) execute the agent in a secure execution environment that can enforce these restrictions. The commercial Java-based systems, such as Odyssey [8], Voyager [29], Concordia [35], and IBM Aglets [17], all cryptographically sign the migrating Java code and then enforce access restrictions with the standard Java security mechanisms, i.e., customized class loaders and security managers [6].

Most research systems provide only partial protection for machines, simply because the research focus is often something other than security (or some other aspect of security). Tacoma [13] provides hooks so that a developer can add their own encryption subsystem (and then use this encryption subsystem to sign migrating agents), but does not provide secure execution environments for all of its supported languages. The Tacoma Too project is experimenting with software fault isolation and security automata as a flexible way to enforce access restrictions [26]. (A security automata is a state machine in which each transition corresponds to an *allowed* resource access; software fault isolation instruments the machine or object code with security checks.) Ara enforces restrictions on CPU time and memory usage, but does not yet protect resources such as the filesystem and network [22]; the Ara group is currently implementing a full security model, however, including digital signatures and access restrictions for all system resources. Both Tube [10] and SodaBot [5] provide execute their agents inside secure interpreters that enforce some access restrictions. (SodaBot agents are written in a custom language called SodaBotl, while Tube agents are written in Scheme.)

D'Agents does focus on security issues and provides relatively complete protection for machines. Agents are cryptographically signed using PGP [15], while access restrictions are enforced with Safe Tcl [18], Java security managers [6], and Scheme 48 modules. Unlike most mobile-agent systems, D'Agents is designed to support multiple languages, and thus focuses on cleanly separating enforcement from policy and on implementing as much of the security mechanisms as possible in a language-independent manner. The D'Agent mechanisms for protecting machines is the focus of most of this chapter.

We plan to use electronic cash to protect groups of machines [2]. Most mobile-agent projects, including Tacoma [13], Ara [22], and Messengers [1], have similar plans. To our knowledge, however, little implementation work has been done by any of these projects.

Finally, there are a range of possible techniques for protecting an agent from malicious machines, most of which were introduced by other mobile-agent projects, and none of which are currently implemented in D'Agents. Since a later section is devoted to describing these techniques, we will present related work in that section.

3 D'Agents

D'Agents is a mobile-agent system whose agents can be written in Tcl Java and Scheme [1] D'Agents has extensive navigation services [23], security mechanisms [9], and debugging and tracking tools [11]. In addition, it is active use at numerous academic and industrial research labs, including labs at Lockheed Martin, Siemens, Cornell, and the University of Bordeaux, and is starting to find its way into production-quality applications.

Like all mobile-agent systems, the main component of D'Agents is a server that runs on each machine. When an agent wants to migrate to a new machine, it calls a single function, `agent_jump`, which automatically captures the complete state of the agent and sends this state information to the server on the destination machine. The destination server starts up an appropriate execution environment (e.g., a Tcl interpreter for an agent written in Tcl), loads the state information into this execution environment, and restarts the agent from the exact point at which it left off. Now the agent is on the destination machine and can interact with that machine's resources without any further network communication. In addition to reducing migration to a single instruction, D'Agents has a simple, layered architecture that supports multiple languages and transport mechanisms. Adding a new language or transport mechanism is straightforward: the interpreter for the new language must support two state-capture routines, and the "driver" for the new transport mechanism must support asynchronous I/O and a specific interface. The primary language is Tcl, and we are currently adding support for Java and Scheme. The primary transport mechanism is TCP/IP.

Figure 1 shows the D'Agents architecture. The core system, which appears on the left, has four levels. The lowest level is an interface to each available transport mechanism. The next level is the server that runs on each machine. This server has several tasks. It keeps track of the agents running on its machine, provides the low-level inter-agent communication facilities (message passing and binary streams), receives and authenticates agents that are arriving from another

[1] See URL `http://www.cs.dartmouth.edu/~agent/` for software, documentation, and related papers.

host, and restarts an authenticated agent in an appropriate execution environment. The third level of the architecture consists of the execution environments, one for each supported agent language. All of our languages are interpreted, so our "execution environments" are just interpreters, namely a Tcl interpreter, a Scheme 48 interpreter, and the Java virtual machine. For each incoming agent, the server starts up the appropriate interpreter in which to execute the agent. It is important to note that most of the interface between the interpreters and the servers is implemented in a C/C++ library and shared among all the interpreters. The language-specific portion is just a set of stubs that call into this library.

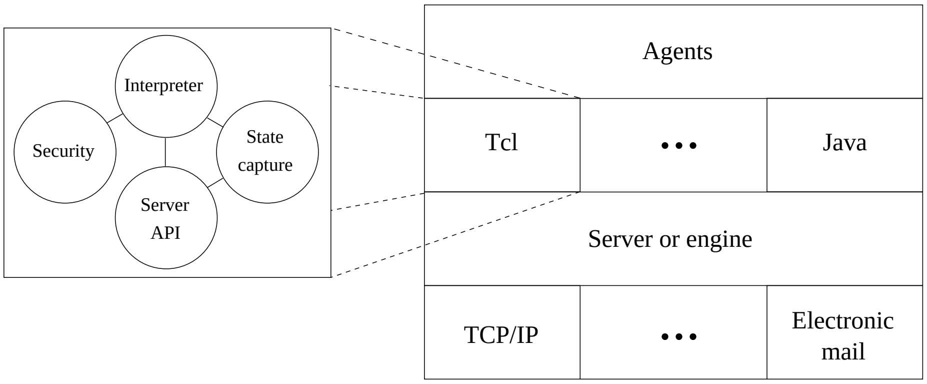

Fig. 1. The architecture of the D'Agents system. The core system, shown at left, has four levels: transport mechanisms, a server that runs on each machine, an interpreter for each supported agent language, and the agents themselves. Support agents (not shown) provide navigation, communication and resource management services to other agents.

The last level of the architecture are the agents themselves, which execute in the interpreters and use the facilities provided by the server to migrate from machine to machine and to communicate with other agents. Agents include both moving agents, which visit different machines to access needed resources, as well as stationary agents, which stay on a single machine and provide a specific service to either the user or other agents. From the system's point of view, there is no difference between these two kinds of agents, except that a stationary agent typically has authority to access more system resources. The agent servers provide low-level functionality. All other services are provided at the agent level by dedicated service agents. Such services include navigation, high-level communication protocols, and resource management.

Figure 2 shows one of the applications in which D'Agents is used. The application's task is to search a distributed collection of technical reports for information relevant to the user's query. The user enters a free-text query into a front-end GUI. The GUI then spawns an agent to actually perform the query. This agent makes two decisions. First, if the connection between the *home* machine (i.e., the user's machine) is reliable and has high bandwidth, the agent stays on the home machine. If the connection is unreliable or has low bandwidth, which is often the case if the home machine is a mobile device, the agent jumps to a proxy site within the network. This initial jump reduces the use of the poor-quality link to just the transmission of the agent and the transmission of the final result, conserving bandwidth and allowing the agent to proceed with its task even if the link goes down. The proxy site is dynamically selected according to the current location of the home machine and the document collections.

Once the agent has migrated to a proxy site, if desired, it must interact with the stationary agents that serve as an interface to the technical report collections. If these stationary agents provide high-level operations, the agent simply makes RPC-style calls across the network (using the inter-agent communication mechanisms). If the stationary agents provide only low-level operations, the agent sends out child agents that travel to the document collections and perform the query there, avoiding the transfer of large amounts of intermediate data. Information about the available search operations is obtained from the same directory services that provide the location of the document collections. Once the agent has the results from each document collection, it merges and filters those results, returns to the home machine, and hands the results off to the front-end GUI for display to the user.

Although the behavior of this agent is relatively complex, it is actually quite easy to implement. Figure 3 shows the Tcl code for a simplified version of the information-retrieval agent. This simplified version always jumps to the proxy site and always spawns child agents, rather than using the network-sensing and directory services. Since using the network-sensing and directory services involves only a few library calls, however, the real agent, including appropriate error-checking and the code to merge and filter the query results, is only about three times as long as the simplified agent.

In addition to excluding error checking, network-sensing and directory lookups, the simplified version of the agent does not explicitly use the D'Agent security services (although it is still subject to the security constraints set by the proxy and collection machines). A version of the agent that does use the security services is presented in the next section.

4 Protecting the machine (and other agents)

Protecting the machine involves two tasks:

- *Authentication.* Verify the identity of an agent's owner.
- *Authorization and enforcement.* Assign resource limits to the agent based on this identity and enforce those resource limits.

D'Agents, like other mobile-agent systems, handles these two tasks with public-key cryptography and secure execution environments that perform authorization checks before each resource access. More specifically, D'Agents has an en-

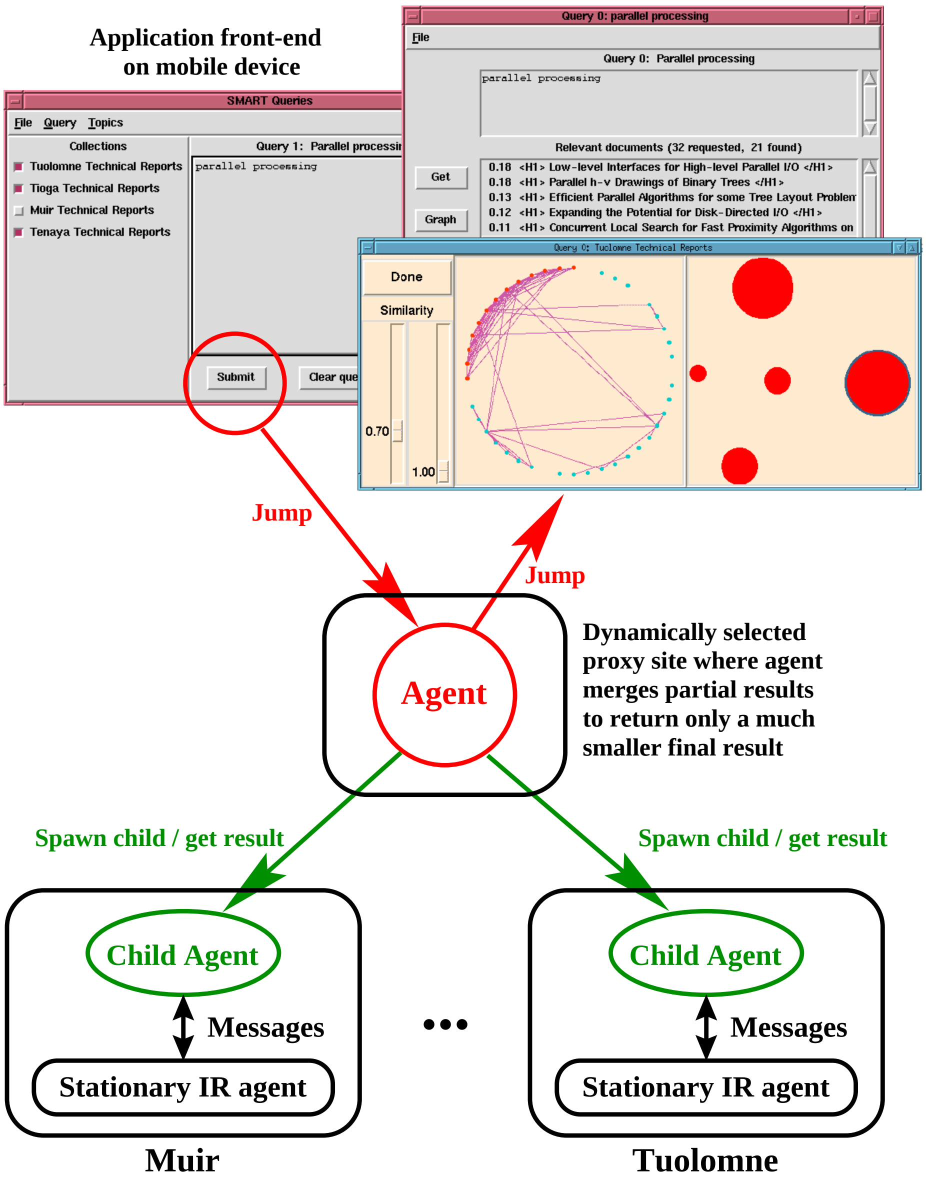

Fig. 2. An information-retrieval application in which D'Agents is used. The user enters a free-text query via a front-end GUI; the GUI then launchs an agent that will search a distributed collection of technical reports for documents relevant to the query. The agent first jumps to a proxy site if the link between the user's machine and the network is unreliable or has low bandwidth. Then, if the query requires multiple operations against each search engine, the agent launches child agents that travel to the search-engine locations and perform the query steps locally to the engine. If the query requires only a single operation, the agent will interact with the search engines remotely.

```
1   proc runQuery {query expansionWords} {
2       global agent
3       # send query to search engine
4       agent_send "$agent(local-server) search-engine" 0 $query
5       agent_receive code results
6       # expand query if we do not have enough results
7       ...
8       return $results
9   }
10
11  # register with the agent system and then migrate to the proxy site
12  agent_begin
13  agent_jump $proxySite
14
15  # send a child agent to each document collection
16  foreach site $collectionSites {
17      agent_submit $site \
18          -vars query expansionWords -procs runQuery \
19          -script {runQuery $query $expansionWords}
20  }
21
22  # receive the query results
23  for {set i 0} {$i < $numSites} {incr i} {
24      set source [agent_receive code results]
25      set queryResults($source) $results
26  }
27
28  # merge and filter the results and then return home
29  ...
30  agent_jump $agent(home-machine)
```

Fig. 3. Tcl code for a simplified version of the information-retrieval agent. First the agent registers with the agent system (line 12). Then the agent jumps to the proxy site (line 13). Once on the proxy site, the agent sends a child agent to the location of each document collection (lines 15–20). The child agents, whose code consists of the `runQuery` procedure (lines 1–9), communicate with the collection search engines to perform the query, and then return their results to the main agent. Finally, the main agent receives the results from each child agent, merges and filters these results, and jumps back to its point of origin where the results are displayed to the user (lines 22-30). (The variable `agent` is a global array that is always available to an agent and that contains information about the agent's current location.) The real version of the agent performs several additional actions that are not shown here, namely (1) appropriate error-checking, (2) using the network-sensing services to decide whether or not to jump to a proxy site, (3) using the directory services to identify the proxy site (variable `proxySite`) and the collection sites (variables `collectionSites` and `numSites`) and to decide whether or not to spawn the child agents, and (4) obtaining the query and expansion words (variables `query` and `expansionWords`) from the front-end GUI.

cryption subsystem, a language-dependent enforcement module, and a language-independent policy module (for each system resource). These three components are shown in Figure 4 and described in the subsections below.

4.1 Authentication

Each D'Agents server distinguishes between two kinds of agents: owned and anonymous. An *owned* agent is an agent whose owner could be authenticated and is on the server's list of authorized users. An *anonymous* agent is an agent whose owner could not be authenticated or is not on the server's list of authorized users. Each server can be configured to either accept or reject anonymous agents. If a server accepts an anonymous agent, it gives the agent an extremely restrictive set of resource limits.

RSA public-key cryptography is used to authenticate an agent's owner. Each owner and machine in D'Agents has a public-private key pair. The server can authenticate the owner if (1) the agent is digitally signed with the owner's public key or (2) the agent is digitally signed with the sending machine's key, the server trusts the sending machine, and the sending machine was able to authenticate the owner itself. In the second case, the sending machine would have authenticated the owner in one of the same two ways: (1) the agent was signed by the owner or (2) the agent was signed by one of the *sending machine's* trusted machines (and that trusted machine was able to authenticate the owner itself). Thus, trust is transitive, and trust relationships must be established carefully. Typically machines under single administrative control would trust each other and no one else.

D'Agents uses Pretty Good Privacy (PGP) for its digital signatures and encryption. PGP is a standalone program that allows the secure transmission of electronic mail and is in widespread use despite controversies over patents and export restrictions [15]. PGP encrypts a file or mail message using the IDEA algorithm and a randomly chosen secret key, encrypts the secret key using the RSA public-key algorithm and the recipient's public key, and then sends the encrypted key and file to the recipient. PGP optionally adds a digital signature by computing an MD5 cryptographic hash of the file or mail message and encrypting the hash value with the sender's private key. Although PGP is oriented towards interactive use, it can be used in an agent system with minimal effort. In the current implementation, D'Agents runs PGP as a separate process, saves the data to be encrypted into a file, asks the PGP process to encrypt the file, and then transfers the encrypted file to the destination server. This approach is much less efficient than tightly integrating PGP with the rest of the system, but is simpler and more flexible, especially since it becomes trivial to create an D'Agents distribution that does not include PGP or that uses different encryption software [30].

An agent chooses whether to use encryption and signatures when it migrates or sends a message to another agent. If the agent is not concerned with interception during migration, it turns off encryption. If the agent is not concerned with tampering during migration and can accomplish its task as an *anonymous*

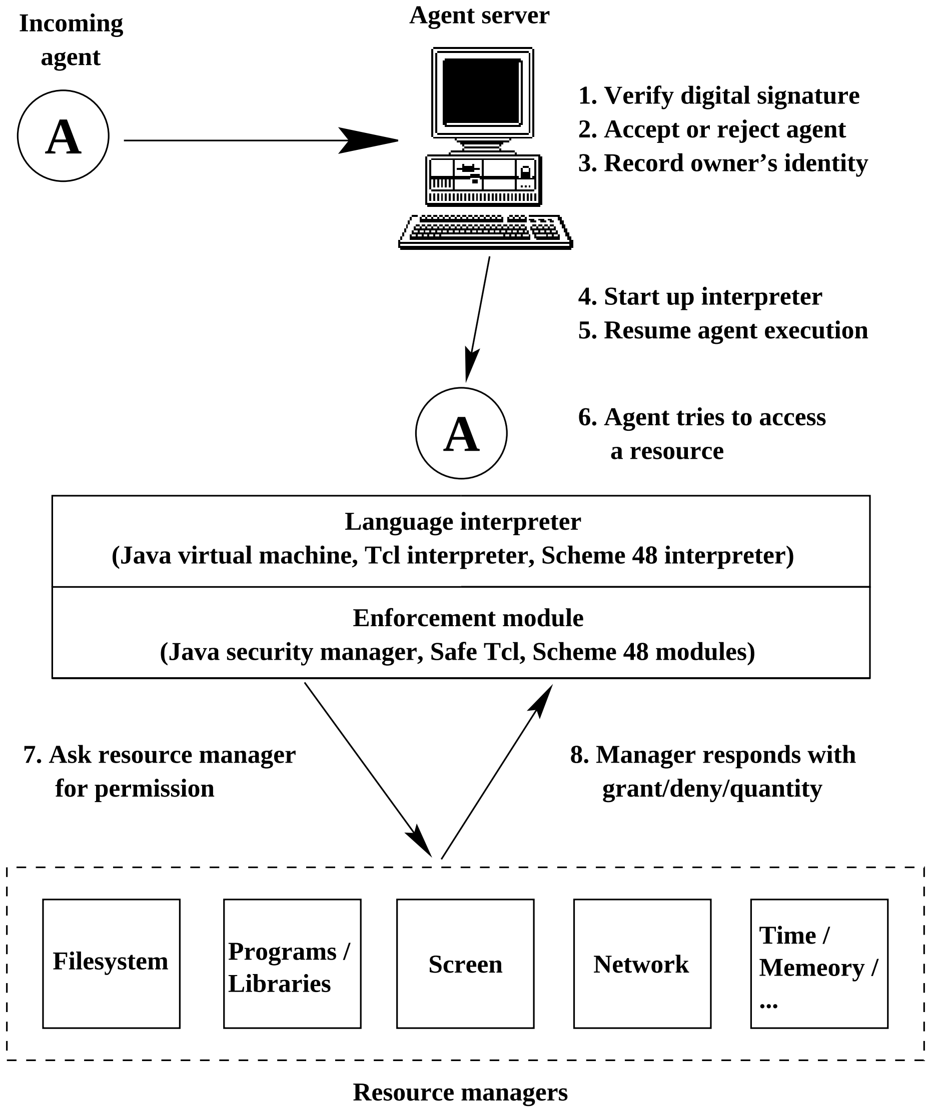

Fig. 4. The components of the D'Agents security architecture. When an agent arrives at an agent server, the server verifies the agent's digital signature if present (step 1), and then either accepts or rejects the agent according to its current access lists (step 2). If the server accepts the agent, it records the identity of the agent's owner for future use (step 3), starts up an execution environment for the agent (step 4), and resumes agent execution (step 5). Once the agent is executing, it might try to access some system resource such as a particular file (step 6). The language-specific enforcement module sends the access request to the appropriate resource manager, which is just a stationary agent that defines the security policy for that resource (step 7). The manager checks the request against its current policy and returns a grant or deny message to the enforcement module (step 8).

agent, it turns off signatures. When sending a message, the agent makes the same decisions, except that it turns off signatures only if the recipient does not need to verify the sender's identity. Turning off either encryption or signatures is a significant performance gain due to the slowness of public-key cryptography, and thus most agents will turn off encryption and signatures whenever the needed resources and the network environment allow it. In the rest of this section, we assume that the agent does *not* want to be an anonymous agent and does *not* want to send anonymous messages, and thus has digital signatures turned on.

When an agent registers with its home server using the `begin` command (Figure 5), the registration request is digitally signed with the owner's private key, optionally encrypted with the destination server's public key, and sent to the server. The server verifies the digital signature, checks whether the owner is allowed to register an agent on its machine, and then accepts or rejects the request. If the agent and the server are on different machines, all further requests that the agent makes of the server must be protected to prevent tampering and masquerade attacks.[2] Ideally, the system would generate a secret session key, known only to the agent and the server, and then use this session key to encrypt the requests [15]. PGP does not provide direct access to its internal secret-key routines, however, making it impossible to generate and use session keys without modifying PGP. Therefore, the current implementation of D'Agents handles the additional requests in the same manner as the initial registration request, digitally signing them with the owner's private key. Since public-key algorithms are much slower than secret-key algorithms, we will switch to secret session keys once we replace PGP with a more flexible encryption library. When the agent and the server are on the same machine (which is the predominant case), there is no need for a session key, since it is impossible to intercept or tamper with the additional requests or to masquerade as the registered agent.[3] Thus all additional requests are transmitted in the clear.

When an agent migrates for the *first time* with the `jump` command, the state image is digitally signed with the owner's private key, optionally encrypted with the destination server's public key, and sent to the destination server. The server verifies the digital signature, checks whether the owner is allowed to send agents to its machine, and accepts or rejects the incoming agent. This process is shown in Figure 6. Of course, once the agent has migrated, the owner's private key is no longer available. Therefore, for all subsequent migrations, the agent is digitally signed with the private key of the sending server. If the destination server trusts the sending server, and the sending server was able to authenticate the owner itself, the destination server considers the owner authenticated and gives the agent the full set of resource limits for that owner. If the destination server does not trust the sending server, or the sending server could not authenticate the owner itself, the destination server considers the agent to have no owner and will either (1) accept the agent as an anonymous agent or (2) reject the agent

[2] A masquerade attack here is another agent passing itself off as the registered agent.

[3] The server uses different communication channels for local agents and can tell without cryptography whether a request came from a specific local agent.

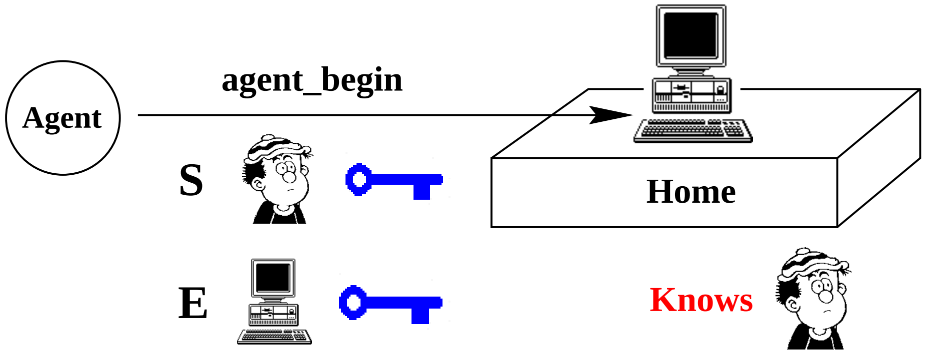

Fig. 5. Encryption for the begin command. When an agent uses the `begin` command to register with the server on its home machine, the registration request is signed with the owner's private key (S) and optionally encrypted with the receiving machine's public key (E).

if it is not allowed to accept anonymous agents. Typically, D'Agents servers are configured so that machines under single administrative control trust each other but no one else.[4] Thus, if an agent migrates from its home machine into a set of mutually trusting machines (and then stays within that set), each machine will be able to (directly or indirectly) authenticate the owner, and will give the agent the full set of access permissions for that owner. Once the agent leaves the set of machines, however, it becomes anonymous, and remains anonymous even when it comes back, since the nontrusted machines might have modified the agent in a malicious way. While the agent is on a particular machine, it will make requests of that machine's server. As in the case when an agent registers with a server on the same machine, however, no encryption or digital signatures are needed for these requests.

When a new child agent is created on a different machine (with the `fork` or `submit` command), or when a message is sent to an agent on a different machine (with the `send` command), the same strategy is used as with `jump`. The message or child agent is signed with the owner's key if the sending agent is still on its home machine, and with the machine's key if the sending agent has already migrated (Figure 7). The recipient server will believe the owner's identity if it trusts the sending server. When receiving a message, the recipient agent gets both the message and a security vector. The security vector specifies the owner of the sending agent, whether the owner could be authenticated, the sending machine, whether the sending machine could be authenticated, whether the message was encrypted, and whether the sending agent is on the same machine. The recipient agent, which might be controlling access to some resource such as a database, bases its own security decisions on this security vector. When a new agent is

[4] For example, all the machines in the Computer Science Department at Dartmouth trust each other.

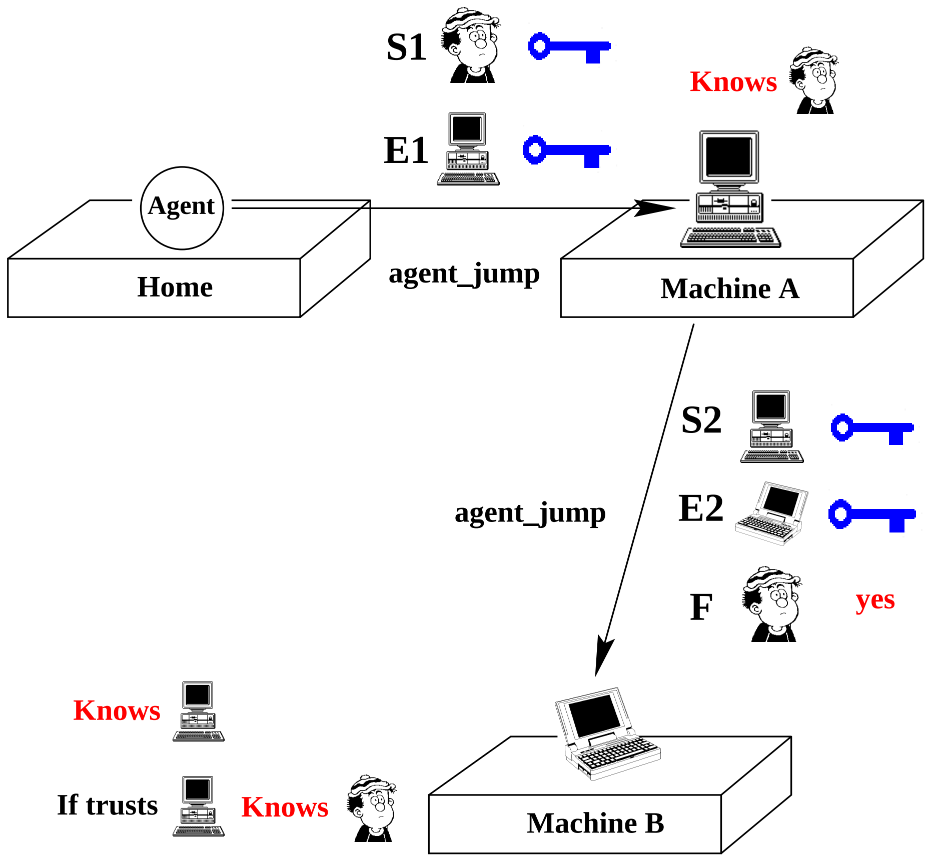

Fig. 6. Encryption for the jump command. On the first jump, the agent is signed with the owner's private key (S1). On the second and later jumps, the agent is signed with the sending machine's private key (S2), and the sending machine sets a flag (F) to indicate whether it was able to authenticate the agent's owner itself; if the target machine trusts the sending machine, and the sending machine reports that it was able to authenticate the agent's owner, the target machine considers the owner authenticated.

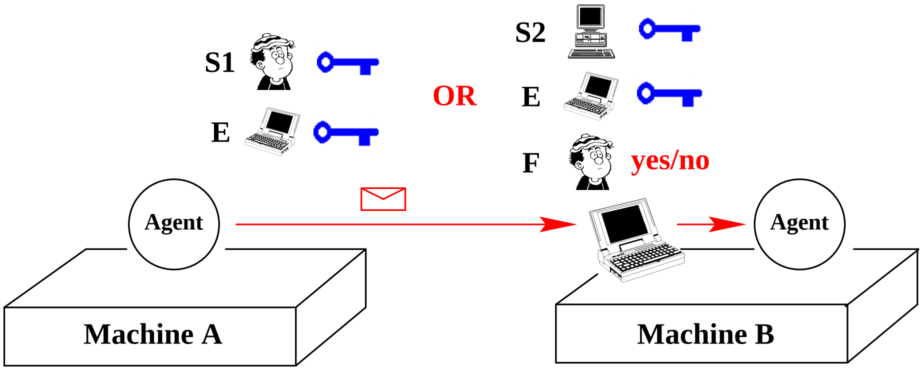

Fig. 7. Encryption for the send command. If the agent has not left its home machine, the message is signed with the owner's private key (S1). If the agent has left its home machine, the message is signed with the sending machine's key (S2), and the sending machine sets a flag (F) to indicate whether it was able to authenticate the agent's owner itself; if the target machine trusts the sending machine, and the sending machine reports that it was able to authenticate the agent's owner, the target machine considers the owner authenticated.

created on the same machine, or a message is sent to an agent on the same machine, no encryption or digital signatures are required. The new agent inherits the security information of its parent. The recipient of the message gets the same five-element security vector.

This authentication scheme has five weaknesses. First, and most serious, once an agent leaves its home group of trusted machines, it becomes anonymous as soon as it migrates again. Making the agent anonymous is essential in the current system since a malicious machine can modify an agent arbitrarily (or lie about the identity of its owner). Thus, when dealing with machines that do not trust each other, an application that needs the full access rights of its owner to accomplish its task cannot send out a single agent that migrates through the machines, since the agent will become anonymous on the second jump. Instead the application must send an agent to the first machine, wait for the results, send a new agent to the second machine, and so on. Although this problem does not prevent an application from accomplishing its task, it places an additional burden on the programmer, and reintroduces some of the network traffic that mobile agents are meant to avoid. At the same time, it is important to note that many applications operate entirely within a set of trusted machines, and that many others, especially in the Internet, can be accomplished with anonymous agents. Solving the multi-hop authentication problem revolves around detecting malicious modifications to an agent. Then, confident that certain kinds of malicious modifications (such as modifications to the static code) will always be detected, a machine can assign access rights that fall somewhere between those of an anonymous agent and those of the actual owner. Detecting malicious modifications is discussed below.

The remaining four problems are less serious and have clear solutions. First, PGP is extremely slow, especially since D'Agents executes PGP as a separate process. PGP must be replaced with a faster encryption library. Second, PGP does not provide access to its internal encryption routines, making it impossible to generate session keys for ongoing communication. The replacement library must support both public-key and secret-key cryptography. Once the system can generate session keys, it should use session keys rather than public/private keys whenever possible due to the speed advantage of secret-key cryptography, For example, two servers that are communicating extensively might generate a shared session key, even if different agents are responsible for each communication. Third, D'Agents does not include an automatic distribution mechanism for the public keys. Each server must already *know* the public keys of all authorized users so that it can authenticate incoming agents (agents signed with an unknown public key become anonymous). A modest key-distribution or certification mechanism must be added to D'Agents to reduce the burden on the system administrator. Finally, the system is vulnerable to replay attacks in which an attacker replays a migrating agent or a message sent to an agent on a different machine. Here a server could have a distinct series of sequence numbers for each server with which it is in contact.

4.2 Authorization and enforcement

Once the identity of an agent's owner has been determined, the system must assign access restrictions to the agent (*authorization*) and ensure that the agent does not violate these restrictions (*enforcement*). In other words, the system must guard access to all available resources. We divide resources into two types. *Indirect* resources can be accessed only through another agent. *Builtin* resources are directly accessible through language primitives (or libraries) for reasons of efficiency or convenience or simply by definition. Builtin resources include the screen, the file system, memory, real time, CPU time, and the agent servers themselves.[5]

For indirect resources, the agent that controls the resource enforces its own access restrictions, rejecting or allowing requests from other agents based on the security vector attached to the incoming communication. Typically, the resource agent would simply check each request against an access list, although one request could return capabilities for use in later requests. Care must be taken with capabilities, however, since a migrating agent will carry its capabilities along with it, possibly through malicious machines. One reasonable solution is to allow an agent to obtain a capability only if it is on the same machine as the resource, and include sufficient identification information in the capability so that it becomes invalid as soon as the agent leaves[6]; this solution makes it impossible for valid

[5] The agent servers are accessed through the agent commands, such as `begin`, `jump` and `send`. All agent commands use server CPU cycles; several use server memory; and several require network access.

[6] For example, the capability could include the agent's id and the time at which it arrived on the local machine. The agent will get a different timestamp (and usually

capabilities to exist on other machines, preventing theft *and* eliminating severe administrative problems. D'Agents will eventually provide both access-list and capability libraries for use in resource agents; currently each resource agent must provide its own implementation.

For builtin resources, the agent servers enforce several absolute access policies. For example, an agent can *terminate* another agent only if its owner is the system administrator or if it has the same owner as the other agent. The *name* operation reserves certain symbolic names for certain agent owners, preventing an arbitrary agent from masquerading as a service agent (such as a yellow page agent that provides directory services). The *notify* operation requires the server to remember which agent asked for the notification, taking up server memory. Thus, the server has a per-agent limit on the number of outstanding notifications; the limit is small for visiting agents, but large for agents that belong to the machine's owner or administrator, since notifications are the most efficient and convenient way to implement monitoring tools that track which agents are currently on the machine.[7] There are similar access policies for the other agent operations. In particular, most operations can be configured to reject requests from remote machines. In a typical configuration, for example, the *begin* operation rejects any request from a remote machine, allowing only agents on the local machine to register with the server. The *begin* operation also imposes a limit on the total number of agents and the total number of *anonymous* agents executing on the machine at one time. The specific limits and access restrictions are specified in a server configuration file.

For all other builtin resources, security is maintained using the language-specific security (or enforcement) module and a set of language-independent *resource-manager* agents. When an agent requests access to a builtin resource, either implicitly or explicitly, the security module forwards the request to the appropriate resource manager. The resource manager, which is just a stationary agent, implements a security policy that determines whether the access request should be approved or denied. The security module then enforces the decision (and also caches the decision when appropriate to minimize the load on the resource managers). This approach provides a clean separation between security policy and mechanism, with the same resource managers making security decisions for all agents, regardless of their implementation language. A system administrator can easily change the security policy by choosing a different resource-manager implementation.

There are currently six resource managers and three enforcement modules (one for each language)in the D'Agents system. Each of them are described below.

id) if it leaves and returns, making it impossible to reuse the capability after a migration. In addition, since the ids are locally unique, no other agent can *ever* have the same combination of id and timestamp, making it impossible to transfer the capability to another agent.

[7] Or, more precisely, notifications will be the most convenient way once an agent can request notifications for a wider range of events.

Resource managers

- *Consumables.* This resource manager handles consumable resources, such as CPU time, wall-clock time, number of child agents, maximum depth of the parent-child hierarchy, and number of migrations. Unlike other resources (such as the file system), access control is not an issue, only allocation. Furthermore, although there is an infinite supply of these resources, each agent should be limited to a finite consumption to prevent system overload. Since access to these resources is either implicit (as with CPU time) or takes place through the generic agent core (migration), enforcement actually takes place in the core, with the language-specific security module simply setting the new limits after the manager returns its decision. In addition, in contrast with the other builtin resources, the agent starts with a small allowance and must *explicitly* ask the manager for more.
  Limits on these resources are enforced across *groups* of mutually trusting machines. When making its decision, the consumables manager considers the amount of the resource used by the agent on other machines within the group.[8]
  Notably absent from this set of consumable resources is memory. Our concern is that a visiting agent could mount a denial-of-service attack against other agents by allocating all available virtual memory (or, indirectly, most available physical memory). A coarse-grained solution is trivial for Java and Scheme 48, which use their own memory-allocation routines and already have command-line arguments to specify a maximum heap size. Tcl, on the other hand, calls the standard `malloc` and `free` routines, and we have not yet implemented the necessary wrappers for these routines.
  Also absent from the current set of consumable resources is *CPU seconds per real second*, i.e., the fraction of the CPU cycles available to the agent, and agent operations per real second. Our concern is that a visiting agent could mount a denial-of-service attack against other agents by sitting in a computationally-intensive loop, or flooding the local server with requests. A solution to this problem requires better support from the operating-system scheduler.
- *File system.* This manager controls read and write access to files and directories. It also imposes a maximum size on writable files so that an agent cannot fill up the file system. Thus it has two roles: access control and allocation. Access control, as in most file systems, is determined on a file-by-file, whole-file basis. (If record-based access control is necessary, the data should be accessed through a stationary database-manager agent.) The main weakness of the current file-system manager is that it does not impose a limit on disk accesses per second, making it possible for an agent to thrash the local disk. Again, we would require more support from the operating system.
- *Libraries.* This manager determines which libraries of Tcl functions, Scheme functions, or Java classes each agent can load.

[8] Migrating agents include a vector that specifies how much of each resource they have used so far.

- *Programs.* The *programs* manager determines which external programs each agent can execute. Since an external program is not subject to the same security checks as the agents themselves, policies implemented by this manager tend to be conservative. Typically, visiting agents obtain necessary services through requests to trusted stationary agents that perform sensitive tasks with careful security checks.
- *Network.* This manager decides which agents are allowed to directly access low-level TCP/IP and UDP network services. It either grants complete access to the network, or no access at all. Again, the policies implemented here is usually conservative.

  We plan to expand this manager's capabilities to allow the manager more flexibility, for example, to distinguish between different hosts, domains, ports, or protocols. In particular, one reasonable policy would allow all agents to access certain RPC-based services, especially when they are on a dedicated proxy site. Then, if a resource is not on an agent-enabled machine, an agent can migrate as close as possible to that machine and interact with the resource using standard cross-network calls [19].

  Again, as with the consumable and file-system managers, we do not currently support usage-rate limitations, such as messages per second. Ideally, some operating-system support would allow us to control access to the network bandwidth.
- *Screen.* Our current screen-manager mechanism can, as with the network manager, allow all access or allow no access. Thus, policies tend to be conservative, disallowing visiting agents any access to the screen. Our concern is that malicious agents might, for example, create a window that covered the entire screen and then grab the global focus.

  We are currently expanding the capabilities of the screen manager to allow more detailed control, making decisions about the number, placement, and size of windows, among other things. Our initial policy will be determined by the user: the screen-manager agent itself pops up a GUI window allowing the user to set limits on each agent that arrives.
- *Others.* There are other resources for which we do not currently have resource managers, such as microphones, speakers, cameras, and printers. They would all fit into the same security architecture, which provides two options. The resource may only be available indirectly, through requests sent to a specialized service agent, or it might be directly available (as with resources like the screen and network) after access has been granted by the appropriate resource-manager agent. The choice is determined primarily by performance considerations.

Security policies Most of our resource managers are currently implemented with extremely simple security policies. Each resource manager has a configuration file that specifies the access rights and limits for a particular *owner*. The manager loads this access list on startup and then checks the owner of each requesting agent against the list. Of course, the manager also takes into account whether the owner could be authenticated and whether the requesting agent is on the same machine. Anonymous agents are given limited access rights (mainly read access to certain libraries and initialization files), and remote agents are given no access rights.

We are currently implementing more involved security policies for the network resource, in which the network-resource manager allocates bandwidth to agents according to the outcome of a competition in which agents bid for access.

Enforcement modules Each language (Tcl, Java, and Scheme) needs its own enforcement module, although as we mention above, some of the resource decisions are enforced by the common code in the agent core. We discuss each of the three languages below, but first we discuss features common to all three.

Decision caching. Since the resource managers are implemented as separate agents, and communication between the visiting agent and the resource-manager agent involves passing messages between processes, we need to keep that communication to a minimum. In particular, it would be too inefficient to ask the resource manager for permission to read each character of a file, to display each pixel on the screen, or to send each packet on the network. Thus, our enforcement modules cache the decision of the resource managers in an internal access list.

For example, if an Agent Tcl program issues the Tcl command `exec ls`, the Tcl enforcement module (see below) checks the internal *program* access list. If permission to execute `ls` has already been granted, the command proceeds. If permission to execute `ls` has already been denied, the command throws a security exception. Otherwise the command contacts the *program* resource manager, adds the response to the *program* access list, and then either proceeds or throws the security exception.

Caching of the resource-manager decisions does not preclude dynamic changes to access-control policy. The caches simply must be invalidated whenever the policy is changed. We are currently working on a graphical administrative utility that lets the machine owner or administrator change the current policies of the resource managers. This utility sends the policy changes to the resource managers and cache invalidation messages to all running agents. Open issues involve sending the cache-invalidation message efficiently and *revoking* a resource permission that has already been granted to an agent. In the latter case, an agent might have opened too many windows, and the user wants to not only change the screen security policy, but also force the agent to close some of its existing windows. Such revocation is quite complex if we allow the agent to continue executing. For example, an agent must include significant error-handling code to handle the sudden disappearance of a window or the sudden closure of a file to which it previously had access. Simpler solutions would be to either terminate the agent or to send the agent back to its home machine.

"Require." An agent can also explicitly ask a resource manager for access permissions with the `require` command. The `require` command takes the symbolic

name of the resource manager, e.g., *filesystem*, and a list of *(name, quantity)* pairs that specify the desired access permissions, e.g., *(/home/rgray/test.dat, read)*. The `require` command causes the enforcement module to send the list of desired access permissions to the appropriate resource manager. The procedure waits for the response and then adds each access permission to the internal access lists, indicating for each whether the request was granted or denied. Regardless of whether an explicit request is made via the `require` command, or an implicit request is made via the use of a sensitive command, the resource manager will send back the most general access permissions possible, effectively preloading the internal access lists and eliminating future requests. For example, if an agent requests access to a particular file, but is actually allowed to access the entire file system, the manager's response will grant access to the entire file system. In addition, although the current implementation does not prevent an agent from contacting the resource managers directly, such contact accomplishes nothing since the response will not go through the enforcement module and will thus not have any effect on the internal access lists.

"Restrict." An agent can impose access restrictions on *itself* with the `restrict` command. In the case of the *consumable* resources, these access restrictions remain in effect even when the agent migrates to a new machine. For example, the agent can restrict itself to a particular number of children, even if it is migrating and creating the children on different machines. More usefully, perhaps, the agent can restrict itself to a specific amount of CPU or wall time.

Tcl enforcement module. The Tcl enforcement module is implemented with Safe Tcl. Safe Tcl is a Tcl extension that is designed to allow the safe execution of untrusted Tcl scripts [18, 21]. Safe Tcl provides two interpreters. One interpreter is a "trusted" interpreter that has access to the standard Tcl/Tk commands. The other interpreter is an "untrusted" interpreter in which all dangerous commands have been replaced with links to secure versions in the trusted interpreter. The untrusted script executes in the untrusted interpreter. Dangerous commands include obvious things such as opening or writing to a file, creating a network connection, and creating a toplevel window. Dangerous commands also include more subtle things such as ringing the bell, raising and lowering a window, and maximizing a window so that it covers the entire screen. Some of these subtle security risks do not actually involve damage to the machine or access to privileged information, but instead involve serious annoyance for the machine's owner.

Agent Tcl uses the generalization of Safe Tcl that appears in the Tcl 7.5 core [18]. Agent Tcl creates a trusted and untrusted interpreter for each incoming agent. The agent executes in the untrusted interpreter. All dangerous commands have been removed from the untrusted interpreter and replaced with links to secure versions in the trusted interpreter. The secure version contacts the appropriate resource manager and allows or rejects the operation depending on the resource manager's response.

The Safe Tcl security module does not provide safe versions of all dangerous commands. For example, an agent that arrives from another machine cannot use the Tk `send` command, which sends a Tk event to another Tk interpreter.[9] In addition, there are (currently) no safe versions of the network and screen commands, since the resource managers either grant complete access to the screen and network or no access at all. The network and screen commands simply remain "hidden" until the resource managers grant access.

Java enforcement module. The Java enforcement module is implemented as a Java security manager [6]. A Java security manager is a class that provides a set of access-control methods, such as `checkExec`, `checkRead`, and `checkExit`. The Java system classes call these methods to see if the corresponding operation is allowed. For example, the `System.exec` method calls `checkExec` to see if the Java program is allowed to execute the specified external program.[10] Our security manager for agents is exactly equivalent to the Safe Tcl mechanism above: each `checkXXX` method checks its internal access list, and if necessary contacts the appropriate *resource manager*; it then throws a security exception if the resource manager denies access. Implementation of the Java security manager is not yet complete. Since the methods follow the same logic as the corresponding Safe Tcl procedures, however, implementation is proceeding rapidly.

Scheme enforcement module. Scheme 48 has a module system [16]. A *module* is a set of Scheme functions with some of those functions marked as *exported* or *public*; a program can load the module and invoke any of the exported functions. Implementing the Scheme enforcement module is mainly a matter of redefining the system modules so that they no longer export dangerous functions, but instead export secure versions of those functions that perform the same security checks as in Tcl and Java. It appears that the necessary module redefinitions can be accomplished without changing the Scheme 48 virtual machine. Implementation of the Scheme enforcement module is also not complete.

4.3 Status

The mechanisms for protecting the machine are nearly complete. There are a few remaining issues, some of which will be resolved soon, and some are left for future work:

– The implementation of the Java and Scheme enforcement modules is expected to be complete in Spring 1998.
– The screen manager and network manager are being expanded to allow finer-grained control.

[9] It is likely that the Tk `send` command will never be available since it is difficult to make secure and agents should communicate within the agent framework anyway.
[10] The *filename* of the external program is a parameter to `checkExec`.

- The current implementation requires that a new enforcement module be written for each language. This approach minimizes the changes to the standard interpreters, but is time-consuming and error-prone. Eventually we will move to the Ara model in which the core provides secure versions of all system functions [22]; these core functions would still contact the resource managers to determine access rights.
- An agent can still mount several denial-of-service attacks: (1) it can sit in a tight loop and consume CPU time as fast as possible; (2) it can flood the local agent server with requests; (3) it can flood the local network by sending requests to remote agent servers as fast as possible (or by using some network service such as RPC to which it has been given direct access); (4) it can allocate all available virtual memory; and (5) it can thrash the local disk by randomly reading from any file to which it has been given access (or by allocating a data structure that is too large for main memory and then accessing the data structure in such a way as to cause frequent page faults). Preventing these denial-of-service attacks is not difficult; preventing them without artificially reducing performance is difficult (and impossible using only our current enforcement modules). Efficient allocation of the available resources to the current set of agents requires more support from the underlying operating system, as well as an appropriate allocation policy. The former is an implementation issue; the latter is an open research question.
- The specification of appropriate security policies, whether in the context of our security infrastructure or another, is a critical area for future research. The Aglets project has one preliminary proposal [14].

Finally, we note that other security models exist. D'Agents uses discretionary access control, in which each resource has an associated access list that specifies the allowed actions for each agent owner. Other security models include (1) mandatory access control, in which programs, people and data are assigned classification levels, and information can not flow from higher to lower levels, (2) security automata [25], in which a program's current allowed actions depend on its past resource usage,[11] and (3) computer immunology [7], in which a program is considered malicious if its current pattern of resource usage does not match its normal pattern. It is an open research question to decide which, if any, of these models is most appropriate for mobile-agent systems.

4.4 Examples

Figures 8 through 11 show two sample agents that use the D'Agent security features. One agent is an information-retrieval agent that jumps to a site, interacts with a search engine to perform some query, and then jumps back to the home machine. The other agent is the search engine itself. Figures 8 and 9 show the agents implemented in Tcl. Figures 10 and 11 show the agents implemented in

[11] For example, an agent might be permitted to communicate with a remote machine as long as it has not read from a sensitive file.

```
1   # turn on digital signatures
2   security signatures on
3
4   # register with the agent system
5   agent_begin
6
7   # migrate to the search engine site
8   agent_jump $engineSite
9
10  # interact with the search engine
11  agent_send "$agent(local-server) search-engine" 0 $query
12  agent_receive code results
13  ...
14
15  # return home
16  agent_jump $agent(home-machine)
```

Fig. 8. Tcl code for a simple information-retrieval agent. The agent registers with the agent system (lines 4–5), migrates to the location of a search engine (lines 7–8), performs a multi-step query (lines 10–13), and then returns home (lines 15–16). (The variable `agent` is a global array that is always available to an agent and that contains information about the agent's current location; initialization of the variables `engineSite` and `query` is not shown.) The only security feature in this agent is line 2, which turns on digital signatures so that the machine `engineSite` can verify the identity of the agent's owner. (Since PGP is slow, the current default is both digital signatures and encryption off; once we replace PGP with a faster encryption subsystem, the default will be digital signatures on and encryption off.) The search-engine agent, which is shown in Figure 9, makes more extensive use of D'Agent security features.

Java. In general, there is a one-to-one correspondence between the Tcl agents and the corresponding Java agents, except that the Tcl agents access agent services through a set of Tcl commands, while the Java agents access agent services by creating an instance of a class `Agent`.

The retrieval agent only uses one D'Agent security feature. It turns on digital signatures so that the search engine's machine can verify the identity of the agent's owner. The search-engine agent uses more D'Agent security features. First, it requests access to the needed system resources, namely, real time so that it can live for a long time, and the filesystem so that it can access the index to the document collection and the documents themselves. Then, after it receives a query from a retrieval agent, it rejects the query unless the agent is on the same machine and has a verifiable owner. It also rejects the query if the owner is not on its own access list. Not all search engines will reject queries in these situations. Some search engines that provide a high-level interface might allow remote queries, whereas some search engines will make their service available to anyone. Such search engines would simply exclude the corresponding security checks.

```
1   # turn on digital signatures and register with the agent system
2   security signatures on
3   agent_begin
4   agent_name search-engine
5
6   # ask for a long lifetime and for access to the document collection
7   require wall $lifetimeSeconds
8   require file [list $documentIndex read]
9   require directory [list $documentDirectory read]
10
11  # wait for queries
12  while {1} {
13
14      # wait for a query
15      set id [agent_receive code string -security secVector -blocking]
16
17      # make sure that the querying agent is on the same machine
18      if {[lindex $secVector 3] != "agent-auth"} {
19          agent_send $id 1 ERROR; continue
20      }
21
22      # make sure that the querying agent has an authenticated owner
23      set ownerInformation [lindex $secVector 0]
24      if {[lindex $ownerInformation 1] != "owner-auth"} {
25          agent_send $id 1 ERROR; continue
26      }
27
28      # make sure that the authenticated owner is on our access list
29      set ownerName [lindex $ownerInformation 0]
30      if {[isAllowed $ownerName] != "yes"} {
31          agent_send $id 1 ERROR; continue
32      }
33
34      # handle the query
35      ...
```

Fig. 9. Tcl code for the search-engine agent. The search-engine agent handles queries from the information-retrieval agents. This version of the search-engine agent will only accept queries from agents that are on the same machine and only from agents whose owner is on a collection access list. After registering with the agent system, the agent requests access to the needed system resources (lines 6–9). Then the agent waits for a query (lines 14–15). Once the agent has a query, it verifies that the querying agent is on the same machine (lines 17–20), that the owner of the agent could be authenticated (lines 22–26), and that the owner is on the collection access list (lines 28–32). As with the other code examples, error-checking and some initialization code (and procedure `isAllowed`) have been omitted for clarity. In addition, note that this search-engine agent must be started by an owner (such as the machine administrator) whose agents are allowed to access the filesystem and to live for a long time. (Also note that the `lindex` command is simply used to access a particular element of a Tcl list.)

```
1   // create the agent
2   Agent agent = new Agent ();
3
4   // turn on digital signatures
5   agent.setSignatures (true);
6
7   // register with the agent system
8   agent.begin ("localhost", timeout);
9
10  // migrate to the search engine site
11  agent.jump (engineSite, timeout);
12
13  // interact with the search engine
14  Message queryMessage = new Message (0, query);
15  AgentId engineAgent = new AgentId (engineSite, "search-engine");
16  agent.send (engineAgent, queryMessage, timeout);
17  ReceivedMessage resultsMessage = agent.receive (timeout);
18  ...
19
20  // return home
21  String homeMachine = agent.getHomeId().getMachine();
22  agent.jump (homeMachine);
```

Fig. 10. Java implementation of the information-retrieval agent from Figure 8. The Java agent first creates an instance of the class `Agent`, which provides all of the agent operations. After that, the Java agent corresponds exactly to the Tcl agent. It turns on digital signatures (lines 4–5), registers with the agent system (lines 7–8), migrates to the location of the search engine (lines 10–11), performs the multi-step query (lines 13-18), and returns home (lines 20–22). As with the Tcl examples, error-checking and some initialization code have been omitted. In addition, the definition of the enclosing Java class (and method) has been omitted, but this class is a normal Java class.

```
1   # turn on digital signatures and register with the agent system
2   Agent agent = new Agent ();
3   agent.setSignatures (true);
4   agent.begin ("localhost", timeout);
5   agent.name ("search-engine");
6
7   # ask for a long lifetime and access to the document collection
8   FilePermission indexPermission =
      new FilePermission (documentIndex, FilePermission.c_READ);
9   DirectoryPermission directoryPermission =
      new DirectoryPermission (docDirectory, FilePermission.c_READ);
10  TimePermission timePermission = new TimePermission (lifeSeconds);
11  agent.require (indexPermission);
12  agent.require (directoryPermission);
13  agent.require (timePermission);
14
15  # wait for queries
16  while (1) {
17
18      # wait for a query
19      ReceivedMessage queryMessage = agent.receive (timeout);
20      AgentId senderId = queryMessage.getId();
21      Security securityVector = queryMessage.getSecurity();
22
23      # make sure that the querying agent is on the same machine
24      if (!securityVector.isAgentAuth()) {
25          ...; continue;
26      }
27
28      # make sure that the querying agent has an authenticated owner
29      if (!securityVector.isOwnerAuth()) {
30         ...; continue;
31      }
32
33      # make sure that the authenticated owner is on our access list
34      if (!isAllowed (securityVector.getOwnerKeyname())) {
35         ...; continue;
36      }
37
38      # handle the query
39      ...
```

Fig. 11. Java implementation of the search-engine agent from Figure 9. Aside from the creation of the initial `Agent` instance, the Java code corresponds exactly to the Tcl code. After registering with the agent system, it requests access to the needed system resources (lines 7–13), waits for a query (lines 18–21), and makes sure that the querying agent is on the same machine (line 23–26) and has an allowed, authenticated owner (lines 28–36). As with the Java retrieval agent, error-checking, some initialization code, and the definition of the enclosing class and method have been omitted.

5 Protecting a group of machines

There are two distinct types of machine groups to protect: (1) all the machines are under single administrative control, as in a departmental LAN, or (2) all the machines are *not* under single administrative control, as in the Internet. The key difference is that machines within an administrative domain typically trust each other, but distrust most other machines in the Internet.

5.1 Within an administrative domain

It is straightforward to protect a group of machines that *are* under single administrative control. An agent is assigned a maximum resource allowance when it first enters the machine group. The allowance and the amount that the agent has used so far is propagated along with the agent as it migrates. If the agent exceeds its group allowance, it is terminated.[12]

The current implementation of D'Agents provides this kind of group protection, by including a *usage vector* in migrating agents. The usage vector lists the maximum allowance and the amount used, for each resource. Agents entering an administrative domain have their maximum allowance reduced if it exceeds that permitted to the agent's owner in that domain. The usage is updated and the limits enforced by the enforcement modules and resource managers described above.

Normally, packet tampering is not a serious issue within an administrative domain, so the agents and the agent's usage vector do not need to be encrypted or signed. If the administrative domain does span suspicious network links, however, each machine must digitally sign the agent and its usage vector so that the usage cannot be reduced, or the allowance increased, during transit. Thus, once an agent is inside an administrative domain, its usage vector is securely maintained and enforced.

5.2 The general case

When the machines are not in a single administrative domain, matters become much more complex. The usage vector carried by an agent migrating between mutually distrusting machines must, for all intents and purposes, be ignored, for the destination machine cannot trust the fact that the source machine has properly accounted for the agent's usage or properly retained the maximum allowances.

A more attractive solution, which we are currently exploring and implementing [2], is to use a market-based approach in which agents pay for their resource usage with cryptographically-protected electronic cash (for example, [3,

[12] Alternatively, the agent could be sent back to its home machine or to a designated proxy site, although the current D'Agents system does not provide such functionality. An agent can inspect its group allowance, however, and can migrate out of the machine group if it sees that it is about to run out of some resource.

27]). When an agent is created, it is given a finite currency supply from its owner's own finite currency supply. The currency does not need to be tied to legal currency, but it must be impossible to spend a currency unit more than once, and it must be impossible for a user to quickly accumulate an arbitrarily large supply. The agent pays for its resource usage with its currency and splits its currency with any child agents that it creates. Eventually the agent and all its children run out of currency and are sent back to the home machine, which either provides more currency or terminates the agent. Resource managers accumulate payments on behalf of the machine's owning user, who can then use the cash to pay for his or her own agents' travels.

There are several advantages to a market-based solution. First, all machines need not trust each other; they need only trust a set of banks that manage the currency. Second, by setting prices accordingly, each machine can express its resource-allocation priorities, e.g., some users may raise the cost of their CPU time during the work day, so that most agents stay away, and lower the cost of CPU time when they are away to allow agents to visit. Third, the agents can autonomously decide how to spend their currency to accomplish their task according to their own priorities, e.g., choosing space-efficient algorithms when memory is expensive, and time-efficient algorithms when CPU is expensive.

There are many detailed issues that must be resolved to make the market-based approach work well, most notably the development of policies for resource managers to set prices, and policies for agents to make decisions about prices. We are examining all of these issues in the D'Agents project, and their full exposition is beyond the scope of this paper.

6 Protecting the agent

Protecting an agent from a malicious machine is the most difficult security problem. Unless "trusted (and tamper-resistant) hardware" is available on each agent server [4], something which is extremely unlikely in the near future, there is no way to prevent a malicious machine from examining or modifying any part of the agents that visit it. Thus, the real problem is not to prevent theft and tampering, but instead to prevent the machine from using stolen information in a meaningful way and to detect tampering as soon as possible, ideally as soon as the agent migrates onto the next machine. Unfortunately, there is no single mechanism that can solve this problem, and it is unlikely that there will ever be a complete *technical* solution, due to the unimaginable variety of theft and tampering attacks that can be mounted against a visiting agent. Instead, some part of the solution will always be sociological and legal pressures [4].

There are, however, several partial technical solutions. Hopefully, by choosing from these partial solutions, most agents will be able to protect themselves adequately for their current task, but still move freely throughout the network. Before considering some of these partial solutions, it is worthwhile to consider two broad categories of tampering attacks.

- *Normal routing.* The malicious machine allows the agent to continue with its normal itinerary, but holds the agent longer than necessary, charges the agent extra money, or modifies the agent's code or state. Holding the agent longer than necessary prevents a time-critical agent from accomplishing its task. Modifying the agent's code or state causes the agent to perform some work on behalf of the malicious machine, take some dangerous action, or simply reach an incorrect result. These modification threats are why D'Agents agents currently become *anonymous* as soon as they migrate through an untrusted machine.
- *Rerouting.* The malicious machine reroutes the agent to a machine that it would not have visited under normal circumstances, or prevents the agent from migrating at all and pretends that it is the next machine on the agent's normal itinerary. The latter attack might be used against an agent that is migrating through a sequence of service providers, attempting to find the best price for some service or product. A service provider can hold the agent on its machine, masquerade as the other service providers, and report higher prices than its own price. Although such an attack requires the service provider to recognize what a particular agent is doing and then update the agent's state as if it had actually visited the other machines, many applications will involve pre-packaged agents that users purchase from the application developers. Recognizing and fooling these well-known agents will not be difficult.

Now, with both theft and tampering attacks in mind, we consider the partial solutions.

- *Trusted machines and noncritical agents.* Note that many agents do not need protection at all, either because they are performing some noncritical task (e.g., an anonymous agent interacting with a free search engine), or because they operate entirely on trusted machines (e.g., an agent that is installing new software on a department's machine). Trusted machines can include not only all the machines in your own department, but also machines belonging to large, well-known corporations, such as America Online, Microsoft, Netscape, and United Airlines.
- *Partitioning.* An agent can migrate through trusted machines only, such as a set of general proxy sites under the control of a trusted Internet service provider. Then it either interacts with untrusted resources from across the network using standard RPC, or sends out child agents that contain no sensitive data and will not migrate again, instead just returning their result. More complicated partitioning schemes can be used if needed. In fact, partitioning can achieve as much client protection as in traditional distributed computing, since the sensitive portion of the agent can always be left on the home machine.
- *Replication and voting.* Tacoma uses a replication and voting scheme to handle malicious machines that either terminate an agent outright or provide the agent with incorrect information [20]. Here, if the task requires a single

agent to visit n services in sequence, the application instead sends out several agents, each of which visits distinct but supposedly equivalent copies of the n services. The agents exchange results after each stage, each agent keeping the majority result. Although this scheme prevents many kinds of attacks, it also has several drawbacks. First, there must be multiple copies of each service[13]; in addition, since the copies might be functionally equivalent but not identical, the agent must be able to handle different interfaces and different result formats. Second, if the agents are spending money to access the services, the user will spend much more money than if a single agent had migrated through a single copy. Finally, the cryptographic overhead is large. Despite these disadvantages, replication and voting schemes will be used in many agents, since they are the only way to handle services that provide incorrect information (assuming that the incorrectness cannot be easily detected). Tacoma also includes *rear-guard* agents that restart a vanished agent.

– *Components.* Perhaps the most powerful idea is to divide each agent into components [4]. Components can be added to the agent as it migrates, and each component can be encrypted and signed with different keys. The agent's static code and the variables whose values never change would make up one component, and would be signed with the owner's key before the agent left the home machine. If a malicious machine modifies the code or variables, the digital signature becomes invalid and the next machine in the migration sequence will immediately detect the modification. In addition, if an agent obtains critical information from a service, it can put this information into its own component. Then the component is signed with the machine's key to prevent tampering, and can even be encrypted with a trusted machine's key (e.g., the home machine or a proxy site) so that other machines cannot examine it. Of course, the agent must return to that trusted machine before it can use the information again itself. Similarly, any code or data that is not needed until the agent reaches a particular machine can be encrypted with that machine's key. For example, an agent might encrypt the bulk of its electronic cash with a proxy site's key, so that it could migrate through untrusted machines without worrying about theft. The agent would return to the proxy site when it needed to spend the cash. Depending on the migration model, this component approach also allows a machine to place greater trust in an agent that has migrated through untrusted machines. For example, if the code to be executed on the current machine is in its own component, digitally signed with the owner's key, and this code does not depend on any volatile variables, the code can be executed with the owner's permissions, rather than as *anonymous*. Finally, components make it easier for an agent to use the partitioning approach above; an agent can leave a particular component behind on a trusted machine, or can create and send out a child agent that includes only certain components.

[13] And the copies cannot be under the control of a single organization. Otherwise all the copies might have the same malicious behavior.

- *Self-authentication.* In most agents, certain parts of the agent's state will change as the agent migrates from machine to machine, such as the variable values and the control information on the interpreter's stack. Although it is impossible to detect all malicious modifications to this state information, it is possible to construct an authentication routine that will examine the state information for any obvious inconsistencies or impossibilities [22]. The authentication routine could also examine the current set of components. Such an authentication routine would be placed in its own component and digitally signed with the owner's key. Each agent server would execute the authentication routine, terminating the agent (and notifying the home machine) if the routine finds any inconsistencies. The authentication routine would run as *anonymous* and would only have authority to examine the state image. Like the components themselves, such an authentication routine allows a machine to place greater trust in an agent that has migrated through untrusted machines.
- *Migration history.* It is possible to embed a tamper-proof migration history inside a moving agent [20]. This movement history allows the detection of some rerouting attacks, particularly if an agent is following a fixed itinerary, and, in combination with additional digital signatures, makes it impossible for a malicious machine to drop an entire component from the agent. The movement history could also be examined inside the authentication routine above.
- *Audit logs.* Machines should keep logs of important agent events so that an aggrieved agent or owner can request an audit from an authorized third-party [4]. The auditor would seek to identify the machine responsible for a theft or modification and penalize that machine appropriately. The exact contents of the audit logs is largely an open question. It is clear that all electronic-cash transfers must be logged, however, so that a machine cannot steal electronic cash without providing the desired service. Of course, a malicious machine can construct a false log, so the auditor must look for log entries that are inconsistent with log entries from other machines, rather than just log entries that explicitly indicate a malicious action. In addition malicious machines can collude in their logging to make an honest, intervening machine *look* malicious. Thus, in some situations, the auditor can impose serious sanctions only after it has observed an apparent attack happening to *multiple* agents (that are following different migration trajectories).
- *Encrypted algorithms.* Finally, recent work [24, 12] involves encrypting a program and its inputs in such a way that (1) the *encrypted* program is directly executable, (2) the encrypted program performs the same task as the original program, and (3) the output from the encrypted program is also encrypted and can only be decrypted by the program encrypter. Although this work is in its infancy and remains either theoretical or unproven, it has great promise for mobile-agent systems, since it would become much harder for a malicious machine to make a targeted modification, i.e., a modification with a known, useful effect, to an agent or its state.

Even taken together, these techniques cannot provide complete protection. In addition, many of the techniques involve substantial cryptographic and logging overhead, forcing an agent to trade performance for protection. Most agents should be able to realize adequate protection through some combination of these techniques, however, while still maintaining reasonable performance. The overriding issue is how to design a protection interface that allows the agent to easily use the desired combination of techniques.

None of these solutions are currently implemented in the D'Agents system.

7 Conclusion and future work

D'Agents is a simple but powerful mobile-agent system that supports multiple languages, namely, Tcl, Java and Scheme, and protects machines from malicious agents with a straightforward security model. It has been used in numerous distributed applications, particularly information-retrieval applications, both at Dartmouth and in external research labs. Several areas of security-related future work remain, however. We must address several denial-of-service attacks, finish the electronic-cash system and develop the market-based control policies, and extend the screen and network resource managers to provide finer-grained access control. We are also continuing to develop information-retrieval applications so that we can experimentally compare mobile agents with other approaches, to better evaluate the feasibility of our security mechanisms and policies. As part of this work, we hope to formally characterize when an agent should remain stationary and when and how far it should migrate. Finally, we are continuing to develop support services, such as a debugger, a hierarchical service index, a docking system for mobile computers, and several network-sensing and planning modules. As this work progresses, D'Agents will be able to realize its full potential and become a convenient, efficient, and secure platform for general distributed applications.

8 Availability

The Tcl portion of D'Agents version 2.0, which is the version of D'Agents described in this paper, is available now on the D'Agents web page[14]. The Java portion will be available by the time of publication. The Scheme portion, which is farther from completion, may be available at the time of publication. D'Agents runs on most Unix platforms. A port to Windows 95 and Windows NT is planned, but the completion date is uncertain.

9 Acknowledgments

Many thanks to Scott Silver, Jeffrey Steeves and Jonathan Bredin for their work on the encryption and resource-management subsystems; to Eric White,

[14] `http://www.cs.dartmouth.edu/~agent/`

David Gondek, Alik Widge, Bill Bleier and Joshua Mills for their work on the Java and Scheme components; to the Air Force and Navy for their generous financial support (ONR contract N00014-95-1-1204, AFOSR contract F49620-93-1-0266, and Air Force MURI grant F49620-97-1-0382); and to all the graduate and undergraduate students who have contributed to the D'Agents (Agent Tcl) system over the past four years.

References

1. Lubomir F. Bic, Munehiro Fukuda, and Michael B. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 29(8):55–61, August 1996.
2. Jonathan Bredin, David Kotz, and Daniela Rus. Marked-based resource control for mobile agents. To appear in the conference Autonomous Agents '98, October 1997.
3. David Chaum and Stefan Brands. "Minting" electronic cash. *IEEE Spectrum*, 34(2):30–34, February 1997. Special issue on Technology and Electronic Economy.
4. David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communications*, 2(5):34–49, October 1995.
5. Michael H. Coen. SodaBot: A software agent environment and construction system. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.
6. Gary Cornell and Cay S. Horstmann. *Core Java*. Sunsoft Press (Prentice Hall), 1997.
7. Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997.
8. *Odyssey: Beta Release 1.0*, 1997. Available as part of the Odyssey package at `http://www.genmagic.com/agents/`.
9. Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the 1996 Tcl/Tk Workshop*, pages 9–23, July 1996.
10. David Halls, John Bates, and Jean Bacon. Flexible distributed programming using mobile code. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 225–231, September 1996.
11. Melissa Hirschl and David Kotz. AGDB: A debugger for Agent Tcl. Technical Report PCS-TR97-306, Dept. of Computer Science, Dartmouth College, Hanover, NH, February 1997.
12. Fritz Hohl. Protecting mobile agents with blackbox security. In *Proceedings of the 1997 Workshop on Mobile Agents and Security*, University of Maryland, October 1997.
13. Dag Johansen, Robbert van Renesse, and Fred B. Scheidner. Operating system support for mobile agents. In *Proceedings of the Fifth IEEE Workshop on Hot Topics in Operating Systems (HTOS)*, pages 42–45, May 1995.
14. Günter Karjoth, Danny B. Lange, and Mitsuru Oshima. A security model for Aglets. *IEEE Internet Computing*, 1(4):68–77, July/August 1997.
15. Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. Prentice-Hall, New Jersey, 1995.
16. Richard Kelsey and Jonathan Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4), 1995.

17. Danny B. Lange and Mitsuru Oshima. *The Aglet cookbook*. 1997. In progress. Selected chapters available at `http://www.trl.ibm.co.jp/aglets/aglet-book/index.html`.
18. Jacob Y. Levy and John K. Ousterhout. Safe Tcl toolkit for electronic meeting places. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, pages 133–135, July 1995.
19. Mobile Agent Facility Specification (joint submissions). Technical report, Crystaliz, General Magic, GMD FOKUS, Internal Business Machine Corporation, and The Open Group, 1997. Response to OMG's Common Facility Task Force RFP3. Draft 5 is available at `http://www.genmagic.com/agents/MAF/`.
20. Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 109–114, September 1996.
21. John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The Safe-Tcl security model. Technical report, Sun Microsystems Laboratories, 1997. In progress. Draft available at `http://www.sunlabs.com/people/ john.ousterhout/safeTcl.html`.
22. Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, volume 1219 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer-Verlag.
23. Daniela Rus, Robert Gray, and David Kotz. Transportable information agents. *Journal of Intelligent Information Systems*, May 1997. To appear.
24. Thomas Sander. On cryptographic protection of mobile agents. In *Proceedings of the 1997 Workshop on Mobile Agents and Security*, University of Maryland, October 1997.
25. Fred B. Schneider. Security in Tacoma Too. In *Proceedings of the 1997 DAGSTUHL Workshop on Mobile Agents*, September 1997.
26. Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the 11th International Workshop on Distributed Algortithms*, September 1997.
27. Marvin Sirbu and J. D.Tygar. NetBill: An Internet commerce system optimized for network delivered services. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*. IEEE Computer Society Press, March 1995.
28. Joseph Tardo and Luis Valente. Mobile agent security and Telescript. In *Proceedings of the 41th International Conference of the IEEE Computer Society (CompCon '96)*, February 1996.
29. Voyager technical overview. ObjectSpace White Paper, ObjectSpace, 1997.
30. Peter Wayner. *Agents Unleashed: A public domain look at agent technology*. AP Professional, Chestnut Hill, Massachusetts, 1995.
31. James E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper, General Magic, Inc., 1994.
32. James E. White. Telescript technology: An introduction to the language. General Magic White Paper, General Magic, 1995.
33. James E. White. Telescript technology: Scenes from the electronic marketplace. General Magic White Paper, General Magic, 1995.
34. James E. White. Telescript technology: Mobile agents. 1996.
35. D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet. Concordia: An infrastructure for collaborating mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (MA '97)*, volume 1219 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer-Verlag.