Mobile Agents: A Survey of Fault-Tolerance and Security.

Jason D. Hartline
hartline@cs.washington.edu

November 19, 1998

Abstract

This survey presents issues and current solutions in the area of mobile agent security and fault-tolerance. We discuss the use of accepted authentication and cryptographic signature techniques applied to the agent system security and privacy requirements. We look at the use of type safe languages in solving the problems of security for the agent's computing environment. We give current solutions to fault-tolerance issues for the agent itself and discuss the tradeoffs involved. Throughout the paper, special attention is given to the agent model as it is affected by certain security or fault-tolerance methods.

1 Introduction.

Agent based computing is a networking paradigm with similar motivations to client-server systems and Remote Procedure Calls (RPC). Important constructs are the agents, the computing environment or host, and the network. An agent is an encapsulation of program, data, and state that can be transferred across the network to remote hosts to be executed. Agents are inherently mobile and can use their mobility to collect data from a variety of different hosts. The host is an agent execution environment where agents are allowed to execute, access host resources, and communicate with local and remote agents. Remote agents refers to agents executing in remote hosts agent environments and local agents are agents executing in the same local environment. The network in this model is in general a heterogeneous wide area network with varying network latency and bandwidth limits that connect many hosts that operate under different administrative control.

1.1 Motivation for agents.

Motivations for agents are explored and analyzed in great detail by Harrison *et al.* [9] and a subset of them will be explained in detail here. The basic agent model allows simple agent based solutions to be implemented for a wide variety of problems in today's networks. The following motivations 1.1-M10 highlight the advantages of using agents to implement networked systems. Throughout this paper we will be referring back to these motivations and how they are affected by using a restricted agent model in exchange for fault-tolerance or security guarantees.

M1: Agents can more efficiently utilize low bandwidth connections because they can do data filtering at the server end and only bring the desired data back to the client. Here, the agent visits the host, makes a request of that host that returns a large amount of data, filters out all but the desired data, and returns to the client.

M2: Agents are useful in solving problems with highly interactive client-server systems. These systems can restructured using an agent based system that will lower the bandwidth of the interaction by moving the computation closer to the data. The agent will be transfered to the host, it will hold a long interactive conversation with the host, and then it will return to the client.

This provides better scalability than traditional network limited distributed systems as the agent's interaction with a host does not pass over the network. The only demands on the network are for the agent to be transferred to and from the host. In RPC or message passing systems many messages must be sent back and forth.

M3: Security for the data transfered from the agent to the host is cheap for agent based applications. An agent must be encrypted and authenticated only once per host it visits while every message in a secure RPC conversation must be encrypted or signed.

M4: Agent communication with host services in the presence of a faulty network can be more reliable than in traditional methods because the agent is executing directly on the host that it is communicating with. Once the agent arrives at the host, the agent can happily communicate with the host and compute its desired results without having to worry about packet recovery and varying rate of service over a network. This communication is reliable without added header and error correcting protocols that exist in client-server networks, thus the overhead in message processing is reduced.

M5: Compared to a wide area network, there is negligible message latency in the communication between an agent and its host or other local agents, thus the round trip times are not the limiting factor in an agent based distributed computation.

M6: Agents are useful in disconnected operation in mobile client (lap top) computing. Mobile clients are characterized by having intermittent connectivity to the network, low bandwidth during connections, low storage capacity, and low processing power. Using agents with mobile clients can provide many advantages. The client can launch an agent to do a computation on its behalf and then disconnect from the network.

M7: In client-server applications where the client have low processing power, agents are useful because they can preprocess the data on the server before returning to the client. In fact, in many cases it makes more sense to do these calculations on the server in the first place because that is where the data resides.

M8: Agents are useful for distributing multi-platform clientware. Due to the mobile nature of agents, agent code needs to be able to execute on many platforms. By extending user interface features to the agents' execution environment, software developers using the agent model avoid the expense of porting their software to run on specific platforms.

M9: Agents can provide semantic routing, freeing the end user from remembering server addresses. Agents have the ability to perform intelligent lookup of host addresses based on the service that the agent is looking for and can roam the network freely in search of a particular service or services. Ideally, the end user will specify a request, possibly a search for this paper, and the agent will consult several online search engines for its location.

M10: Agents also provide a means for implementation of an emerging electronic marketplace [25]. It is easy to see how hosts can be made to represent shopping centers and agents will shop and barter for goods on the user's behalf. Agents' ability to filter through big databases on the host and their ability to roam from host to host make them well suited for shopping for items or services.

These motivating advantages that agents provide are inherent in the nature of agents as defined in the previous section. While traditional networking paradigms can be specialized on a per application basis to match these motivations, mobile agents provide a good, clean solution for all of them [9].

1.2 Requirements for security and fault-tolerance in agent systems.

While agents provide system engineers with a new and useful network computing model with many mo-

tivating advantages, agent computing also introduces many security and fault-tolerance issues (1.2-I8):

I1: Agents must not be able to contaminate the host that they are running on with viruses.

I2: Agents must not be able to consume unmoderated amounts of resources and thus effect a denial of service attack.

I3: Hosts must be able to govern the agents ability leak private information outside the hosting system.

I4: Agents must be able to save and restore their state and local variables [1].

I5: Agents need to be persistent through host crashes and premature termination by a Byzantine failed host[2].

I6: Agents need to be able to complete their algorithms they implement correctly and produce accurate results in the face of Byzantine failure of the host execution environment.

I7: Agents should be able to communicate privately with both local and remote running agents without being spied on by their execution environment.

I8: Agents should be able to keep their previously gathered data private and secure from their current host execution environment as well as from host that might intercept the agent in transit.

Note that while I7 seems impossible to solve in an agent based system, the corresponding problem in a client-server system of allowing clients to communicate with other hosts and other clients without risk of being overheard by a particular "cheating" host is easily solvable with public key cryptology. Others of these issues such as I4 and I1 are not even relevant in traditional client-server systems. Issues such as I6 can be guaranteed; however, the cost of guaranteeing the veracity of the result reduces to that of checking the correctness of a computation which in a naive implementation could involve re-simulating the agent and comparing its execution to an execution log provided by the host execution environment. This approach wastes more resources than could be saved by using agents instead of a client-server technique because the entire execution log would have to be transported back to the agent's origin.

Agents without security guarantees are not very attractive. No system administrator would agree to let untrusted executables run on their system without guarantees that their system could not be subverted. Likewise a client would be hesitant to send an agent out into the electronic marketplace to buy goods on the client's behalf if the agent could be tricked into making undesirable purchases. Ultimately, we would like to be sure that the data that the agent returns is the correct data corresponding to a successful, unhampered execution of the agent's code implementation.

1.3 Issues in implementing security and fault-tolerance in agents.

Given a particular security or fault-tolerance goal, we would like a solution that does not restrict the agent model "too much". We would like a solution that does not incur excessive overhead in computation time, network bandwidth, or other resources. We also do not want to limit the functionality of the individual agents, especially in the motivating areas where agent solutions are natural.

Here "too much" and excessive overhead are terms relative to a non-agent implementation or limitedly secure agent implementation of the same application. We need to decide if using agents to perform the task at hand with the given security requirements is more efficient than performing the same task using traditional communication and traditional security techniques.

Robert Morris emphasized in his talk at the Dartmouth Agent Workshop that security should be viewed from an economic standpoint of risk and cost [7]. In cases where the cost of ensuring security or fault-tolerance is very high, the system developers must decide whether that level of fault-tolerance or

[1] Why this is a security issue is explained in section 3.1.1

[2] A host that maliciously tries to interfere with agent computation is Byzantine failed.

security is really necessary. For example, in some cases the cost of fault-tolerance, I6, is high to the point of duplicating computation to verify its correctness. Yet other aspects of a secure and fault-tolerant system can be realized to an acceptable degree with minimal overhead (I1, I2, I3, *etc.*).

It is important to carefully weigh the motivating advantages presented in Section 1.1 with knowledge of the expense involved in attaining desirable security and fault-tolerance guarantees. Agent computing introduces several new failure modes. The cost of providing fault tolerance to a particular agent model might make it undesirable to replace a client-server application with an equally secure agent based system.

1.4 Organization.

We will discuss authentication and integrity checking techniques for the host, agent, and agent origin in Section 2. In Section 3 we will discuss the use of existing techniques for preventing viral attacks and denial of service attacks. Section 4 contains several proposals for solving the problem of fault-tolerance in agent computing. Issues relevant to agent computing in type-safe and restricted languages are discussed in Section 5. Some examples of security systems implemented in selected commercial and experimental agent based systems are given in Section 7. Lastly, our conclusions are given in Section 8.

2 Authentication and integrity.

As with other network protocols, authentication is an issue in agent based computing. A host would like to make sure it knows the origin of an agent before it accepts it for execution. Likewise, an agent might only want to execute on a specific host or set of hosts. The agent and the host need to be able to prove their identities to each other, and the agent must be able to prove that it and its data have not been tampered with.

2.1 Agent integrity and authentication.

An agent's integrity is at stake if it passes over untrusted network connections. A hostile host might intercept and modify an agent as it might any other network packet. To detect illicit modification of the agent, the agent code and data can be digitally signed using existing public key technology.

2.2 Desirability of agent authentication and integrity checks.

Agent authentication and integrity are required in some applications. An agent that comes from a reputable or known source can be given access to more resources on a private server than its unauthenticated siblings [18, 1]. This requires authentication of the agent source and verification that the code in the agent body has not been modified by any other host. For example, agents that originate within a certain company should be allowed to access that company's databases and print servers; however, agents from competing companies should not be able to access confidential information or restricted devices when executing on the same servers.

On the other hand, Harrison *et al.* [9] point out that hosts do not always need or want authentication guarantees. Some host might run free public read-only services and not need to authenticate browsing agents. If all agents are going to be admitted, there is no need for authentication or integrity checks. In some cases where the agent's origin needs to be kept confidential, anonymity of the agent's source is even desirable. Examples of this are search engines that guarantee the privacy of the queries' contents.

2.3 Agent privacy and host authentication.

Agent privacy is also an issue. In some cases data that the agent carries is sensitive, and it would be detrimental if it were intercepted by a hostile third party. This privacy can be granted at a cost. Using well developed public key encryption techniques, the agent body and data can be encrypted with its

destination's public key. This has the obvious implication of limiting the agent to executing on a specific host and thus providing host authentication as well as agent privacy. However, in the case that the information the agent carries is sensitive to interception, it is probably desirable for the agent's execution to be limited to one host.

2.4 Desirability of host authentication.

In many cases in network systems, authentication of the host is critical for maintaining security. If an agent does not authenticate the host on which it wants to execute, the service that that host provides could be spoofed by another host that intercepted the agent for malicious reasons; as well, sensitive data could be intercepted if it is not encrypted.

It is not always the case that host authentication is desirable. Agents, by keeping a list of hosts that are entitled to execute them, limit their ability to roam across the network as required by M9 and this itinerary control is in many cases undesirable [2].

2.5 Integrity of gathered data.

Another issue is the privacy and integrity of collected data, I8. Information gathered by an agent might be sensitive and need to be kept private. For example, a shopping agent that visits several hosts trying to negotiate a lower price for some item has the dilemma that if the host can read the agent's previously found lower price from the agent's private data then it can just state a new lower price just below the previously stored lower price. In addition, this information must not be altered unexpectedly in visits to subsequent hosts. For this, Chess *et al.* suggest using either a *stateless* or *stateful* model [2]. In the stateless model, the agent must communicate gathered data back to its origin as it is gathering it. In an extreme case, after each interaction with a host, the host will encrypt and sign the data on behalf of the agent and send it across the network to the agent's origin. One key point to note about this model is that it limits the agent's origin from disconnecting from the network, M6.

In the stateful model, the host encrypts or digitally signs the data and appends it to the agent's body as it sends the agent on to another host. In this manner, the agent builds up a history of results and then returns to its origin with them.

There are several issues closely linked to this method. First, the server appending the data can not only sign it, but can also digitally encrypt it to keep the information secret from other hosts that are subsequently visited. This prevents the agent from accessing that data as well which limits the agent's ability to complete its desired task and perform intelligent filtering, M1.

If the host cannot read the previously found prices from the agent's state because they are kept in encrypted form then the agent cannot access the gathered data either. The shopping agent will, therefore, not be able to make executive decisions based on its price history. The end user cannot specify that the agent report home when it has found three sites selling widgets for under $50. If the agent were to keep this data and even the instructions "take the first three widgets under $50 and return home" in its state as well as previously found low widget prices then that state information could be read by the host and an adverse strategy could be designed to fool the agent into doing something undesirable.

The other issue here, as mentioned by Chess, is that a malicious host could remove these pieces of gathered data even if they are encrypted or signed. To combat this, they introduce an *audit trail* that would allow a retracing of the agent's itinerary [2]. The audit trail would consist of a list of signed handshakes for each time the agent was exchanged between hosts. As each host passes the agent on to the next host in its itinerary list, the host would add the name of the host to which it was transferring the agent. When the agent finishes its computation at a host, the host would sign all the data previously collected including the handshakes and send the agent on to the next host using secure handshaking. This new host will make sure that it is the intended receiver of the agent and then it will run the agent. When the agent returns to the origin, the origin can verify the signatures and make sure that nothing is missing from hosts visited. If any data is missing

it will be obvious which host removed it because all signatures preceding that of the faulty host will be unreadable and all signatures following it will be correct. This is because all hosts sign the entire contents of the data; if one host is missing then none of the data preceding it can be verified. This does limit the agent's mobility because it restricts the agent to moving in a linear fashion from one host to another. It does not directly support divide and conquer algorithms and the agent's itinerary must be decided upon in advance.

3 Host security.

When a host is executing an agent's instructions it must take precautions so as not to open itself to hostile attack from malicious agents. The host needs to protect itself from viruses, denial of service, and privacy attacks.

3.1 Viral attacks.

It is impossible to verify that an arbitrary piece of code does not contain a virus [3, 2]. However, as Cohen mentions, virus spread can be limited or stopped by restricting the computing environment. In particular Cohen places limitations on the exchange of data across information boundaries by using limited sharing. He also mentions that the same effect can be gained by going to non-Turing models of computation. Harrison *et al.* note that a Turing computational model can be used for the agent programming language; however, it must be that the virus *spread* function is not implementable [9]. In this manner, agent environments can be constructed so as not to allow the host system to be compromised. The Java Runtime [4] and Safe-Tcl Interpreter [6, 1, 18] are examples of restricted computing environments that, in theory, do not compromise the Turing computability model and do offer security from viral attacks while providing a portable programming environment as well (Section 5.1 and 5.2). Proof carrying code can also be used to protect the host from viral attacks. The agent code must be accompanied by a proof that the code is not a virus (Section 6.1). This code binary executable and does not have to be interpreted to insure safety. Thus, there are several methods for protecting a host from viral attacks from agents (I1).

3.1.1 The problem with call stacks.

One requirement in any agent implementation is that agents have to be able to capture their state and move from one host to another (I4). Methods exist in various languages to capture variable contents and the heap storage into a contiguous block of data. However, languages also hold state information in their stack. If agents are allowed to restore their stacks from some sequence of raw data there is a potential to break language enforced security such as that imposed by Java. If we can assume that all hosts are trusted and that a host is responsible for encoding and decoding agent stack information, we can allow the stack to be captured in this manner. However, in a wide area network where hosts are operated under different administration, agents in systems that rely on strict typing for security cannot safely be allowed to restore their stack from data passed from one host the the next because a corrupt host could illegally construct a stack that would allow an agent to break its security restrictions.

This presents a major problem in designing agent based systems. Current Java based implementations [24, 21] require that all state information be kept in the agent's variables or heap storage and that the agent be able to return to that execution state from the contents of its variables (Section 7.4). All stack information must be rebuilt on arrival at a new host from the contents of the agent's variables. To make this possible the agent cannot move between sites without first finishing all partial computations and getting to a state that is representable by data in its local storage. If a host wants to initiate the agent's movement, it must give the agent notification to finish all these partial computations before agent can be transferred. This method poses the burden of maintaining this state information on the programmer and is undesirable for that reason.

3.2 Denial of service attacks.

One crux of denial of service attacks, I2, stems from the fact that it is undecidable whether a given piece of code will exit or not [3]. This model extends to a piece of code that might exit but not before it forks one or more copies of itself to be run on the same or a different host machines. The current solution to this problem is a currency based method [25, 9]. In this method an agent will be given a certain amount of currency which corresponds to how much total execution time or amount of a resource it and all of its forked children are allowed. Eventually an agent and its children will run out of currency and no longer be allowed to run or access system resources.

3.3 Information theft.

Information flow in agent systems must be monitored and restricted to prevent information theft [18] as stated in I3. This mandates that untrusted agents that are allowed access to sensitive information not be allowed to communicate that information to the outer world or to other agents that can communicate with the outer world. We can look at the transitive closure of information flow assuming that entities trusted by the host, such as host originating agents and the host executing environment itself can be trusted not to leak information out of the system and are thus dead ends in the closure. It must be that untrusted agents either are allowed to communicate with the outside system closure or they are allowed to talk to the inside system closure but not both. Once an agent is given permission to view sensitive information it must not be allowed to leave the system or be allowed to communicate with any off system entity or untrusted agent that is allowed to leave the system. An added complication is that there might be different levels of information that need protection. Safe-Tcl provides *security policies* [18] to implement highly configurable restricted environments (Section 5.2). Security policies in the Safe-Tcl system cannot be composed safely. This means that an agent must be given only one security policy.

4 Agent fault-tolerance.

For an agent system to be completely fault-tolerant, it must guarantee that agents and their results are not compromised by the remote host that they are executing on (I6). It must also guarantee that agents do not get prematurely terminated (I5). For the purpose of distinguishing these cases we will associate Byzantine failure with the compromise of the agent's result and crash failure with the premature termination of an agent. Therefore, the Byzantine failure model that we will adopt includes the host removing portions of the agent's code and running other code in its place, the host incorrectly performing the agent's computations, the host exposing the agent's algorithm to other entities in the host, and the host modifying the agent's state inappropriately. The crash failure model includes premature termination of an agent from a malicious host or crashed host and from loss in transit in the network.

We would like agent based systems to be able to cope with crash failure as well as Byzantine failure. Depending on the agent model, several methods have been suggested for fault-tolerant agent computing. When it is feasible to duplicate host services, a system of replication and Byzantine agreement has been suggested by Schneider and is being implemented in the TACOMA project [7, 12, 16]. Another suggested solution is moving agent-agent interactions to a neutral third party host [15]. Yet another suggestion is to use physically secure coprocessors also called tamper-resistant hardware [2, 26]. Return validation and legal protection [26] also provide a solution to the fault-tolerance problem where applicable. Each of these proposed methods work well in solving a subset of our goals given a limited agent model; however, none of them are general enough to be used in every case.

4.1 Duplication of service.

In this scenario we assume that it is feasible to duplicate services. An agent will pass from one stage of services to the next and will do Byzantine agreement at each stage. To insure a correct computation sequence in lieu of Byzantine failure, a stage with

t faulty hosts must have at least $2t + 1$ duplicated hosts. An agent is propagated from one stage to the next and Byzantine agreement is performed taking the answer to be the first $t + 1$ agreeing results. One major advantage in this technique is that if fewer than t hosts fail, the computation need not await these hosts results. The next step can proceed as soon as $t + 1$ matching results arrive.

If a particular host server is running slow or there is a network partition, the computation speed is not limited by the slowest host. Minskey *et al.* also explain how to counter attempts from failed hosts to spoof votes with a method of cryptographic secret sharing. The problem here arises when more than t hosts from different stages try to spoof hosts in the last stage into thinking that they are part of the previous stage [16].

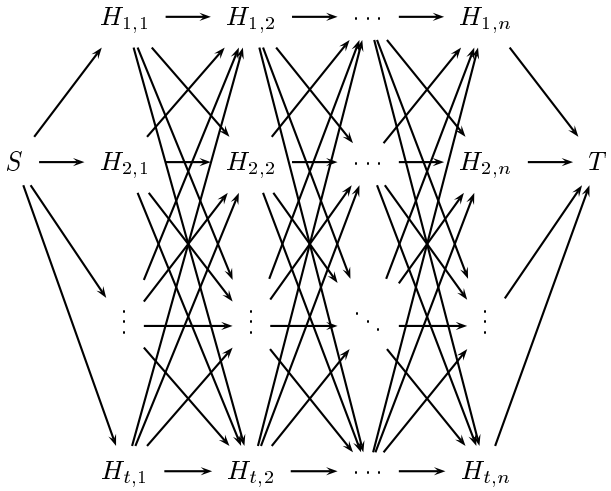

Figure 1: Duplicated service.

If we just assume crash failure and not a Byzantine failure mode, we can have a model in which we do not have to actually run the agent on duplicate hosts until we detect that the first host has been compromised by a crash. In crash failure we need $t + 1$ replicated servers to tolerate t crashes per stage [7].

It is questionable as to whether replication of services is an appropriate solution for Byzantine failure in agent computation because hosts with the same service would likely be under the same administrative control and would potentially all be Byzantine. Schneider proposes a model of approximate service replicas and approximate voting to alleviate this situation. Thus, agents can visit hosts from organizations with similar services and successfully do Byzantine agreement [7].

4.2 Third party rendezvous.

In the cases where there is a small amount of data at a host site that an agent would be interacting with, the computation could, instead of sending the agent to the host, request for the host to send a representative agent to a trusted third party meeting place where the client's agent would meet with the host's agent to negotiate a result [15].

4.2.1 Duplicated third party rendezvous.

This idea of a third party rendezvous is a good one; however, it assumes that a third party host that is trusted by both parties exists and is available. This idea can be modified and extended to protect against third party sites that could be conspiring with one of the agents. In this case, when two agents, A and B, want to interact, they each suggest a third party site that they trust to provide them with a fair computation. Call the site A suggests S_1 and the site B suggests S_2. Each agent will then clone itself and send one copy to each of S_1 and S_2. At S_1, agent A_1 will meet with B_1. At S_2, agent A_2 will meet with agent B_2. Two duplicate computations will follow and then the agents will return to their respective origins to compare answers.

Note that this method cannot protect against disclosure of private information in A or B to each other and the world if either is compromised by the third party site. This is an agent security issue, I8, that seems unsolvable [9].

This situation excludes agents from using the model, M1, that they need to extract a little bit of special information from a huge database. Obviously, it is impractical to send the entire database to a third

party host, and yet M1 is one of the driving motivations for agents.

4.3 Return validation.

Sometimes, as with authentication, a system will not need or want explicit general purpose fault-tolerance guarantees. Some computations are inherently easy to verify and in this case, a general fault-tolerance approach would yield a more computationally or physically expensive system. For example, all NP-complete problems are relatively easy to verify. The answer or *certificate* is the proof and it can be verified in polynomial time. This situation is also known as $\mathcal{IP}(1)$ for an interactive proof with one message passed from the prover, the remote host, to the verifier, the agent's origin [10].

4.4 User enforced fault-tolerance.

When the repercussions of errors in the result of the agent program are not serious or the agent is part of a highly user interactive system where a user will be able to realize if an agent has been compromised, no fault-tolerance needs to be built into the agent system.

4.5 Legal Protection.

In some situations the hosts or servers that agents run on can be guaranteed to execute the agent code as written and not violate the agent's privacy because of a legally binding contract. In such a contract the server operators would guarantee that the server will not compromise the privacy and correctness of the agent's computation. This method does not limit our agent model or compromise any of our motivations; however, as Yee points out [26], for this method to work, there must be a way to detect a breech in contract. Such a breech in contract must be provable to provide evidence for court hearings. If methods for proving a breach in contract existed, then we should be able to solve the security problems without having to resort to legal methods in the first place. In addition, from a computer scientist's standpoint, this is hardly an acceptable solution.

4.6 Secure coprocessors.

Secure coprocessors can also provide a solution to agent fault-tolerance and privacy problems. There have been several implementations of secure coprocessors. Citadel [19] and mABYSS are two that include a general purpose processor, non-volatile memory, cryptosystems, and a mechanism for destroying secret information stored in the non-volatile memory [23]. Systems like these can be used in host computers as the agent processing environment.

A secure coprocessor for agents could be modeled as a black box in which you can input agent code and data that has been encrypted with the black box's public key. The black box would then decrypt the agent and run it. When the agent finishes its execution it can be re-encrypted with the black box's private key and the agent's next destination's public key. The result can be given back to the host operating system to be sent on to its next destination.

In order for this or any solution to be useful for agent based systems, it must be accepted and implemented throughout the agent network. Current cost due to lack of demand and difficulty in constructing tamper-resistant hardware is a major impediment to secure coprocessors being implemented as the solution to agent fault-tolerance. A detailed description of the steps taken to construct a secure coprocessor are given in Tygar and Yee's account of Dyad [23].

The techniques used in mABYSS and Citadel involve building a self-destruction mechanism for erasing all non-volatile storage, especially the private keys, and a system for detecting tampering that will trigger the self-destruction. They use an alarm system that consists of a dense wire winding around the processor that, when broken, triggers self-destruction. This is then encased in epoxy that is chemically stronger than the metal wires so that a drilling attack will result in the breaking of a wire, and a epoxy dissolving attack will result in the dissolving of the wires as well. Also, the processor has a temperature sensor that will erase its memory if an attempt to freeze the processor and its current state is made. As is evident, any system for securing a coprocessor must be complete to the extreme. All attacks must result in self-destruction.

Using secure coprocessors is just a partial solution. The secure hardware must either be hardwired or hardcoded with the agent's execution environment. User customization of this hardware cannot be allowed. Upgrades of the agent execution environment can be distributed in encrypted form from the manufacturer of the secure coprocessor. We must assume that the manufacturer of the coprocessor is honest and does actually provide a fully secure system with no loopholes.

While this solution does provide good privacy and fault-tolerance protection for agents it suffers from the problem that for it to be usable it must be widely accepted with minimal additional server cost. The Sanctuary project [26] is looking into using secure coprocessors in an agent based system.

4.7 Non-volatile storage and a rear guard.

As of yet, all of these techniques assume a Byzantine failure model. Addressing the crash failure model, we need to make sure that agents do not get lost when their hosts crash. We also need to make sure that if an agent is lost in this manner, it can be recovered. Non-volatile storage for the agent state information is one method that will allow the agent to survive host crashes [25].

The idea of a *rear guard* [16, 11] can be used to protect an agent from being terminated prematurely by crash or Byzantine failure. The idea is that every agent will have an associated rear guard agent, a clone of itself, that will be left on the last site that the agent visited to make sure that the agent does not die on the host on which it is currently executing. This can be used in conjunction with trusted hosts that do not implement good crash failure protection. If the machine recovers after a crash without knowledge of the agent, the rear guard agent will re-send the agent to the host. This technique is very similar to the message buffering in network protocol layers used to insure that a message reaches its destination.

The rear guard method adds overhead to the system in bandwidth and storage requirements and it does not successfully prevent the agent from being lost when the two hosts (the one that the agent is currently visiting and the host on which the rear guard resides) both crash. This is not an issue in TACOMA [16] where the idea for rear guards was first suggested because it was assumed that the agent computations were replicated in stages and at each stage most of the hosts had not failed (Section 4.1).

Rear guards and non-volatile storage of agent state are useful in agent systems despite some complications. They provide limited extra fault-tolerance without restricting the agent model too much.

5 Language based security models.

In this section we will discuss language based security models as they are used by agents. Issues in language based security that are not unique to agents will not be discussed here. Several popular languages have built in security models; notably, the Java programming language and the Tcl scripting language. These languages both have the property that they are system independent. Tcl scripts can be run on any machine with a Tcl interpreter and Java executables can be run on any machine that runs the Java virtual machine. This portability as well as their inherent security make these two languages attractive for use with agent systems. It is important to realize that while these languages do provide facility for security, it is up to the implementation of the host environment to correctly take advantage of this security and not leave holes in its interface.

5.1 The Java type-safe system.

Java insures that the host cannot be compromised by running Java executables. This secure system is implemented through strong type checking at compile time and in bytecode verification and overflow checking at runtime as the bytecodes are interpreted. Java has no concept of pointers and does not allow arbitrary type casts. Security for agent systems could be attained much like that of the *applet* system currently in use on the World Wide Web. Security for untrusted applets received from remote machines is attained by limiting the applet's access to file input

and output on the host machine as well as limiting network connections to only the applet's origin. In this manner, the Java runtime environment is secure from viral and privacy attacks [4].

Java also has object *serialization* facilities [5, 24] for saving the heap storage associated with a Java object. This serialization does not include stack state information because, as mentioned in Section 3.1, being able to recreate a call stack from arbitrary data would compromise the language's type system. Instead, the serialization facility allows an object that implements the *Serializable* interface to be written to a stream and extracted from a stream. The class implementation has the responsibility of making sure that all the data in its variables are consistent. Encryption can also be built into the serialization by the object's implementation. It is clear that serialization also provides a means of bypassing the type system; however, this cannot be used to bypass security barriers because not all objects can be serialized and because the object's serialization procedures can be defined by that object. In this respect, agents using serialization can only break the typing of their locally defined classes and not the classes in the host interface. They cannot violate the encapsulation of objects other then their own. A host must simply not allow its interface objects to be serialized without being encrypted or checksummed first. This requirement does not limit the agent because there is no need for an agent to bring a host's implementation objects to the next site that it visits.

The Java applet system that has been popularized by the World Wide Web is not an agent based computing system in the full sense of the term. In fact, the role of client and server are reversed in that the applet origin is the server and the host that runs the applet is the client. The significance of Java to agent systems is that it provides a good starting point to implement such systems. It has a fixed set of library routines and a wide user base making it an ideal choice for agent based systems as is demonstrated with IBM's Aglets [24] and the University of Stuttgart's Mole project [21] (Section 7.4).

5.2 The Safe-Tcl restricted instruction set.

Safe-Tcl started out as a language based on Tcl to allow Enabled Mail by using the `application/Safe-Tcl` MIME type as proposed by Borenstein and Rose [1]. The motivation was to provide a secure environment in which to execute emailed scripts without fear of compromising the system to viral or privacy attacks. This Safe-Tcl provided two interpreters, the untrusted interpreter in which untrusted scripts could be safely executed in a limited environment and the trusted interpreter where scripts could execute with the full set of Tcl builtin functions. The idea was to provide a highly restricted language for untrusted scripts by removing all the commands from the Tcl language that were deemed dangerous. Access to functionality could be provided by allowing the user who owns the mail box to make extension libraries available to be loaded by the untrusted scripts to allow them special restricted access to dangerous builtins.

A version of Safe-Tcl based on that of Borenstein and Rose has made it into the official Tcl releases [18]. The official Safe-Tcl provides a master trusted interpreter that can oversee the execution of untrusted scripts in a safe interpreter. This is a *padded cell* approach that allows a trusted script to isolate the execution of several untrusted scripts as well as providing restricted access to dangerous functions by an *aliasing* mechanism similar to that available in Borenstein and Rose's Safe-Tcl.

The functions accessible though aliasing are prescribed by the master interpreter based on a *security policy* requested by or made available to the untrusted script. Since security policies cannot generally be composed, many security policies can be made available and untrusted scripts must choose only one of these to use for the entirety of their execution [18]. The untrusted script requests a certain security policy and the master interpreter either allows or disallows that policy based on the untrusted script's authentication status or other information.

Tcl with the Safe-Tcl untrusted interpreter is embeddable and extensible making it easy to use as an agent execution environment. Builtin commands can

be added in traditional languages like C and the aliasing feature provides a facility to implement any desired security policy optionally using cryptographic authentication [18]. Safe-Tcl is used as the primary agent implementation language in Dartmouth College's Agent Tcl project [6].

6 Safe binary agents.

Agents can also be transmitted between hosts in precompiled machine code. However, the machine code must be accompanied by a proof or set of annotations that show that it will not harm the host computer. This has the advantage of allowing agents to be executed quickly without an interpreter, and it still prevents the host from being compromised. Again we will be discussing how these methods relate to agent based systems. For an agent based system to use such a system, all hosts in the system must have the same architecture and the union of their security policies must be satisfiable by agent code. Recent work with proof-carrying code [14] and with an annotation based system [13] are given below.

6.1 Proof-carrying code.

Recent research in proof-carrying code has provided a new means for agents to be safely executed on a host. The code is transfered in binary form along with a proof of the code's adherence to the destination host's safety policy.

The proof-carrying code system works as follows. The agent is compiled on the client machine. The client compiler also generates a proof that the agent satisfies a given security policy. The binary code and proof can then be sent to the host machine. The host machine validates the proof and then can safely execute the agent. The agent must be revalidated at each host it visits (under different administrative control). Proofs and code generated in this system have the property that if anyone tampers with the code or proof in transit then on receipt at the destination host either the proof will not validate the code and will be rejected, or it will validate the code, but the code will not compromise the host. The whole system relies on a correct implementation of the validator and a safety policy that does not have loop holes. Current implementations of proof-carrying code are for the DEC Alpha with safety policies written in first order logic. A typed lambda calculus (Edinburgh Logical Framework) is used to generate and validate the proofs [14, 17].

6.2 Annotated binary code.

Another method that is not as expressive as proof-carrying code uses a compiler generated annotations. These can be created as the code is compiled from a type-safe language and can be checked quickly against the binary code in a verification process at the destination host. This system is much like proof-carrying code, but does not involve full blown proof generation and checking. The code generated is more restricted because it has to fit into a less expressive annotation system. It guarantees control flow safety, memory safety, and stack safety. The program cannot jump to or read data from arbitrary places in memory and the code must preserve its call stack. This system is currently being tested with a scheme–like language [13].

7 Security and fault-tolerance implementations in existing agent based systems.

Selected notable existing agent systems are General Magic's Telescript system, TACOMA, Agent Tcl, Mole, IBM's Aglets (using Java), and Sanctuary. Their creators have chosen to address different security issues in their different implementations. Java and Telescript are commercial products that are actively being used on the internet. Agent Tcl, Mole, and TACOMA are research projects that have been tested on the network but have not been adopted by the public. The Sanctuary project is still in preliminary stages and is the only implementation that uses the secure coprocessor to get fault-tolerance and agent security guarantees.

7.1 General Magic's Telescript

The Telescript product [25] is an object oriented agent computing language developed by General Magic, Inc. It is currently being used with AT&T's PersonaLink network [22] and in several of General Magic's own products.

7.1.1 An overview of Telescript.

Telescript distinguishes *places, agents, travel, meetings,* and *connections* as the primitives of the mobile computing environment.

places Telescript places are agent execution environments where agents execute, communicate, and meet with other agents. These places interpret the Telescript code and provide a crash failure secure environment for the executing agent with regard to its temporary variables and execution state.

agents Telescript agents are object-oriented programs written in the Telescript language that can travel from place to place, communicate across the network, and meet with other agents.

travel With Telescript's **go** primitive, agents can request to move themselves from their current Telescript place to another execution place. If the agent is accepted for execution at the destination place then the agent code will resume execution with the command after the **go** invocation.

meetings Agents executing at a Telescript place may request to meet with other agents at that same place with the **meet** primitive. Meeting requests may be denied or accepted by the solicited agent. If a meeting is agreed upon then the two agents may exchange objects.

connections Agents may also communicate across the network using an interface to existing standard communication methods. They may send objects over the network to remote agents with which they are communicating.

7.1.2 Security facilities of Telescript.

The Telescript environment imposes several security facilities. By the interpreted nature of the Telescript code, it can protect the host from virus attack. The Telescript language lacks the capability to let a Telescript agent directly access memory, file system, or other physical resources. In this sense, untrusted Telescripts are limited to computational models that do not support Cohen's *spread* function and are therefore secure from virus infection.

Additional ability to limit the execution model of the agent are given by Telescript's notion of *credentials* and *permits.*

credentials All Telescript agents and places are identified by cryptographic credentials. Agents must prove their credentials upon entering a place from another untrusted place. If an agent moves between two places that are under the same administrative control, there is no need to re-establish the agent's credentials.

permits Permits give an agent limited access to resources such as CPU time, memory, the network, and private data. When an agent enters a new place it negotiates with that place what its permits will be. The place will deny the agent entry unless it agrees to the place's restrictions. The place can only restrict the capabilities of the agent, it may not extend them. The agent permits might include a maximum lifetime, a maximum data size, a maximum expenditure of resources and a maximum allowance (measured in *teleclicks*).

.

In this model, the Telescript places are safe from agents. Agent authentication is effected with a cryptographic system for verifying agent credentials. The agent execution environment is safe from viral attacks because of the interpreted and restricted nature of the Telescript language and it is safe from denial of service attacks because of permits and teleclicks.

7.1.3 Telescript fault tolerance.

Telescript does offer fault tolerance from crash failures. All Telescript agents, their execution states, and local objects running in Telescript places in an execution environment are kept in non-volatile storage as described in Section 4.7. If the server hosting the execution environment fails due to a crash, upon reboot all agents will resume as they were before the machine crashed [25].

7.1.4 Conclusions on Telescript.

The Telescript technology, is a well thought out secure and complete agent computing system. It explicitly deals with many of the issues presented in Sections 2 and 3 without limiting the agents in their mobility and usefulness. Their system; however, does not try to solve the Byzantine failure problem as discussed in Section 4.

7.2 TACOMA.

The TACOMA (Tromsø And COrnell Moving Agents) system defines an agent based system with abstractions of *agents, places, briefcases, folders,* and *file cabinets.*

folders Folders are named objects that contain a list of elements in raw data form.

briefcase Briefcases are containers that each agent owns. Briefcases contain folders and are used for communication. For example, an agent might **meet** another agent by presenting that agent with a briefcase. The briefcase in this example might contain a message for the other agent.

file cabinets File cabinets are local storage units associated with a *place.* This way, agents can communicate with agents that are not yet at that place. For example, agents implementing a divide and conquer algorithm can mark places already visited by leaving a folder in a file cabinet at that place.

The TACOMA system uses a firewall type mechanism between the agent execution and the host system to protect the host from malicious agents. In this manner agents do not need to be implemented in a safe language. Current support is available for implementing agents in Tcl/Tk, C, or Java. TACOMA is exploring agent fault-tolerance through replication as mentioned in Section 4.1 and is implemented over the Horus [20] system. TACOMA also incorporates the idea of *rear guards* [11] as discussed in Section 4.7.

7.3 Agent Tcl.

Agent Tcl, a product of Dartmouth College, provides an agent execution environment that allows agents implemented in secure languages like Safe-Tcl and Java to transport themselves over the network and communicate with local and remote agents. Security is accomplished through the use of these safe languages along with resource managers which grant access to restricted functions based on agent authentication. Safe-Tcl, the main implementation language, has been modified to keep much of the call stack information in local variables so that they can be bundled with the agent's state and sent over the network when the agent migrates.

Agent migration is implemented through a secure public key encryption method which guarantees that agents cannot be intercepted and that they are from the host that they claim to be from. They do not, at this time, protect the agent from being tampered with by each host it visits. It is planned to incorporate Chess's *audit trail* scheme for detecting data tampering as well as the aforementioned currency mechanism for resource allocation [6].

7.4 Java agent implementations: Aglets and Mole

The The University of Stuttgart's Mole project [21] and IBM's Aglet project [24] both use the Java language to gain host security. They use the Java serialization facility to transfer agent state from host to host and require the programmer to manually store all relevant information that would normally be in the stack (which cannot be transferred) as discussed in Section 5.1.

Agents in IBM's Aglet system are called *aglets.* Aglets are an extension of the applet idea where ob-

jects derived from aglets can overload certain creation, destruction, and transport functions. The Aglet system uses a callback method for notifying individual aglets that they need to prepare themselves to be transported to remote hosts. Aglets are allowed to interact through proxy objects. These are required because the creation and destruction functions are public and if a direct reference was made available to other aglets then they could call these functions as well.

An agent in Mole is the transitive closure of objects that reference each other. To communicate with other agents or the host environment agents make use of symbolic references similar to aglet proxies. Straßer *et al.* mention the problem of dealing with multiple threads in a single agent when some of them want to migrate and some do not; however, they leave this unsolved for the programmer to deal with.

Neither Mole agents nor aglets are provided any form of fault-tolerance guarantees. They do, however, provide an agent protection from other agents executing on the same host.

7.5 Sanctuary.

The Sanctuary project [26] at the University of California at San Diego aims to provide a secure infrastructure for mobile agents using a secure coprocessor to provide security and fault-tolerance. The project will provide a system for Java based agents to run on unmodified Java interpreters. Sanctuary will also develop a trust model using public key encryption for use with interserver communication.

One good point about the Sanctuary project, as a secure coprocessor implementation, is that it does not require widespread use of a fixed architecture secure coprocessor. Its use of the Java virtual machine will allow it to be ported to other secure architectures without requiring modification to agent code. In addition, the same Java agents that can run on its secure coprocessor can run at their own risk in environments that do not have the physical security guarantees.

8 Conclusions.

It is important to keep in mind, when designing secure and fault-tolerant systems for agents, that the usefulness of the agent could be undermined and the motivating reasons for using agents for a particular problem could be lost due to inefficiencies incurred by the security system.

Much research has been done in access control, authentication, and integrity verification in other areas of networking which can also be applied to agents. Restricted execution environments for agents can be constructed to make secure agent hosts that cannot be subverted by malicious agents.

The capturing of the agent's state poses a problem for agent systems that rely on type-safety because the agent's call stack cannot be safely reconstructed on a given host. This poses the inconvenience to the agent implementer that they must explicitly keep track of the agent's state.

We also see that there are a number of proposed solutions to special cases of the agent fault-tolerance problem, but there has yet to be a good all around solution that does not restrict the agent in its usefulness, efficiency, or choice of hosts. In fact, it may be that a good all around scheme of fault-tolerance for agents is impossible [2, 6, 15].

References

[1] Nathaniel Borenstein and Marshall T. Rose. MIME Extensions for Mail-Enabled Applications: application/Safe-Tcl and multipart/enabled-mail. Working draft. 1993.

[2] David Chess, B. Grosof, Colin Harrison, David Levine, and Colin Paris. Itinerant Agents for Mobile Computing. IBM Research Report, RC 20010, IBM Research Division, March 1995.

[3] Fred Cohen, Computer Viruses: Theory and Experiment. *Computers & Security* 6, Elsevier Science. 1987.

[4] James Gosling, and Henry McGilton. The Java Language Environment: A White Paper. Sun Microsystems. 1995.

[5] Java Object Serialization Specification. Beta Draft. Sun Microsystems. December, 1996.

[6] Robert S. Gray. Agent TCL: A flexible and secure mobile-agent system. In *4th Annual Usenix Tcl/Tk Workshop*. 1996. URL: `http://www.cs.dartmouth.edu/~agent/papers/tcl96.ps.Z`

[7] Robert S. Gray, Brian Brewington, and Sumit Chawla. Dartmouth Agent Workshop: Summaries of Workshop Presentations. November 1996.

[8] Robert S. Gray. Department of Computer Science, Dartmouth College, Hanover, NH. May 1995.

[9] Colin Harrison, David Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? IBM Research Report, RC 19887, IBM Research Division, October 1994.

[10] Oded Goldreich, Probabilistic Proof Systems - A Survey. Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Israel. 1996.

[11] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating System Support for Mobile Agents. In *The 5th IEEE Workshop of Hot Topics in Operating Systems.*

[12] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Supporting Agents in a Heterogeneous, Faulty and Insecure Network. *15th ACM SOSP*, Colorado, USA. December 1995.

[13] Dexter Kozen, Efficient Code Certification, Tech. Report, Cornell U. January 1998. URL: `http://www.cs.cornell.edu/kozen/papers/cert.ps`

[14] Peter Lee and George C. Necula, Research on Proof-Carring Code for Moble-Code Security. Proceedings of the Workshop on Foundations of Mobile Code Security, Monterey, 1997. URL: `http://www.cs.cmu.edu/~necula/fmcs97.ps.gz`

[15] Anselm Lingnau and Oswald Drobnik. An Infrastructure for Mobile Agents: Requirements and Architecture. Frachvereich Informatik (Telematik), Johann Wolfgang Coethe-Universität, Franjfurt am Main, Germany.

[16] Yaron Minskey, Robbert van Renesse, and Fred B. Schneider. Cryptographic Support for Fault-Tolerant Distributed Computing. Department of Computer Science, Cornell University, Ithaca, NY. July 1996.

[17] George C. Necula, Proof-Carring Code. Presented at POPL97, January 1997. URL: `http://www.cs.cmu.edu/~necula/popl97.ps.gz`

[18] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. The Safe-Tcl Security Model. Sun Microsystems Laboratories, Mountain View, CA. 1997.

[19] Elaine R. Palmer. An Introduction to Citadel - A Secure Crypto Coprocessor for Workstations. In *IFIP SEC'94 Conference*, Curacso, Dutch Antilles, May 1994.

[20] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A Flexible Group Communications System. Dept. of Computer Science, Cornell University. 1995.

[21] Markus Straßer, Joachim Baumann, and Fritz Hohl. Mole - A Java Based Mobile Agent System. Institute for Parallel and Distributed Computer Systems, University of Stuttgart, Denmark, October, 1996.

[22] Telescript Security. Byte. October 1994. URL: `http://www.byte.com/art/sec7/art3.htm`

[23] J. D. Tygar and Bennet Yee. Dyad: A system for Using Physically Secure Coprocessors. Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

[24] Bill Venners. Under the Hood: The architecture of aglets. *Javaworld* . March, 1997. URL: `http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html`

[25] James E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper, General Magic, Inc., 1994.

[26] Bennet S. Yee. A Sanctuary for Mobile Agents. Computer Science Dept. University of California, San Diego. 1997.