Secure Object Spaces

Jan Vitek

jvitek@cui.unige.ch, Object Systems Group, University of Geneva, Geneva, Switzerland

Mobile software agents are computational entities acting on the behalf of a user which may move from computer to computer over a heterogeneous network, draw on local resources, and interact with other agents. This extended abstract discusses agent communication and its implications for security.

1 Introduction

The rapid growth of computer networks has created the opportunity for distributed trading of data and services between loose communities of heterogeneous machines running a variety of software bases and part of different organizations with their own security policies. The challenge is to design a reliable and efficient infrastructure for such communities: trustworthy enough to be a basis for electronic commerce applications, and, at the same time, flexible enough to allow new functionality and upgrades to propagate in the system without any centralized control. Traditional distributed systems are not adapted to the task. Instead we advocate the use of *mobile object systems* (also called *mobile software agents*). A *mobile object system*, or *mobile agent*, is a self-contained, autonomous, collection of objects executing one or more threads of control in order to fulfill a task on the behalf of a user (or *principal*) which may suspend its computation and migrate to another execution context. Agents execute on *agent execution platforms* (AEP), the AEPs provide the run-time support and is in charge of security and access control.

If this technology is to be used for electronic commerce, it must protect all players of these future electronic markets. Mobile object systems require a large and rather complex software infrastructure. The principal security issues (listed in Fig. 1) are: (1) the communication between platforms (i.e. migration of agents) must be secured, (2) incoming agents must be authenticated and granted some rights on the basis of

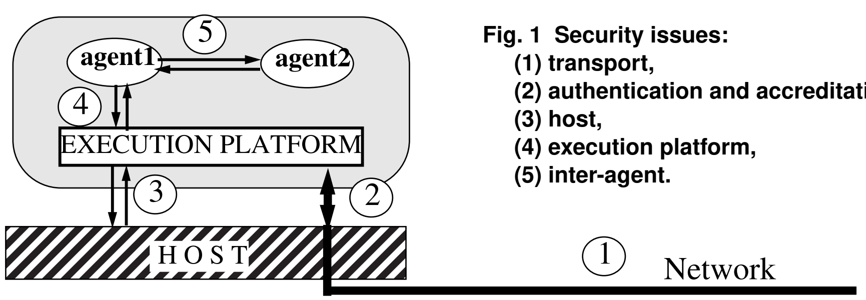

Fig. 1 Security issues:
(1) transport,
(2) authentication and accreditation,
(3) host,
(4) execution platform,
(5) inter-agent.

security	danger	defence
Network	access to private information encapsulated in an agent, tampering with it's state	cryptographic techniques, secure channels, authentication
Host	unauthorized access of local resource, e.g. reading or writing to permanent storage	access to the host granted by the AEP on a discretionary basis
Platform	interfering with the AEP's state by way of reading, writing, or executing code	secure agent-AEP communication and accounting of resource usage
Agent	interfering with the state of an agent by reading, writing, or executing code	secure inter-agent communication and resource usage accounting

their identity, (3) all interaction between the AEP and the host computer must controlled and verified, (4) the AEP must be protected from potentially malicious agents, (5) agents must be protected from each other. Fig. 1 summarizes the risks. We review security issues and asses them briefly. Security of network communication and authentication is a well studied problem often solved by cryptographic techniques. Protection of the host and access rights (issues 2-3) bear strong similarities to protection issues in operating systems. The difficulty is to define which resource to protecte and to provide flexible access control mechanisms. Protection of the AEP has been studied in the framework of executable content languages [12][79], and can be achieved by imposing some restrictions on programs (e.g. forbidding arbitrary memory manipulation). Other approaches to AEP protection use techniques similar to the address space protection in operating systems [89]. The last issue, agent communication, is more troublesome, as agents must be allowed to communicate freely, yet should not be allowed to interfere with each other's execution. This working paper addresses secure inter-agent communication by proposing a new communication and resource accounting mechanism called *secure object spaces*. The concepts presented here are part of a new programming language called SEAL.

1 Security and Communication

This work shifts the attention from protecting the host to protecting agents. Our motivation is that, while security of the host machine and environment is important, agent security is even more important as the agents carry out economic transactions and encode user knowledge. Their privacy must be preserved and their execution must proceed unhindered. We distinguish four categories of attacks:

attack	description
Breach of privacy	direct access to the state of another agent either due to a failure to enforce proper security or an insecure communication channel.
Breach of integrity	modification of the state of an agent by another agent by sending state-modifying messages to objects of the victim.

Masquerading	usurping the authority/identity of another agent for a series of actions, e.g. tricking the victim into executing some code fragment.
Denial of service	excessive consumption of a finite shared resource such as processing time, memory, or the communication subsystem.

1.1 Execution model

A *system* consists of a set of hosts, a set of agent execution platforms, and a set of principals. A host may run zero, one, or more agent execution platforms. Each principal has different access rights depending on host and platform. Hosts, platforms and principals may added or removed at any time. For any object o, the object graph $g^*(o)$ is the transitive closure of all objects reachable from o by following chains of instance variables. An agent $\alpha = (g^*(o), o, p, T)$ consists of a connected graph of objects $g^*(o)$ rooted at some object o, a principal p, and a set of threads T. Every object in $g^*(o)$ belongs to α and every instruction executed by some $t \in T$ is executed on the behalf and with the authority of the principal p. A platform executes a set of agents which may belong to different principals.

1.2 Using message passing for agent-level communication

In an object-oriented setting agents may be represented by objects, thus naturally, agent communication becomes message passing. This allows agents to exchange references to objects and send messages to these objects. While simple, this approach raises difficult security questions: How to guarantee integrity? Who owns an object? For instance, an object o may belong to an agent α but execute on the behalf of an agent β. The memory taken by the object is charged α's account, but authority and processing time are those of β. This complicates the implementation. Also, objects have to check the origin of messages before answering them. If a message originates from an object executing on the behalf of an unauthorized agent, the receiver may decide not to answer it. This approach forces objects to be responsible of their own protection.They must enforce a security boundary at their interface. Proponents of this approach argue for the use of two standard mechanisms of object-oriented languages for security, namely, encapsulation and strong typing. In a way this is conceptual overloading as two concepts, originally designed to enforce good software engineering practice, are put to use towards enforcing security. These are very different goals, it is not surprising that this approach has some weaknesses. Section 1.3 presents several attacks. This approach also has the disadvantage that the software developer must make sure that all objects used in one application implement a coherent security policy and that these objects have been coded carefully so as to avoid security leaks. This is a heavy burden as it implies that security code is spread over many classes.

1.3 Attacks

Breach of privacy

Privacy may be breached each time an object grants access to part of its state to an object of another agent. By a sequence of message it may be possible, starting at an object o, to get to many, or all, of the objects in the data closure $g^{*}(o)$. Thus, a single shared object may open a large portion of an agent's state to inspection. The only appropriate protection is to check in the authorization of the caller before returning any value. That is, the object must have a method for determining which agent has the right to access which parts of its state. All classes involved in inter-agent communication must include checks[1]. There is a more subtle way to breach privacy. It is sufficient to retain a reference to an object given to an other agent. This reference can then be used later to get at the state of that agent. This attack can not be prevented by the typing rules nor by encapsulation.

Breach of Integrity

To protect itself against breach of integrity, all objects involved in agent-level communication must check the identity of the caller (or rather the identity of the agent on whose behalf it is acting) before executing any method which writes to an instance variable [82]. A partial solution is to provide a read-only parameter passing mode [27].

Masquerading

Strong typing and encapsulation can not fully protect an agent against masquerading attacks. For example an attack can take advantage of the type system, more precisely of subtyping, to inject code into an agent. This is achieved by calling a method which requires an argument of some type and passing it a subtype which overrides some methods to perform malicious actions. Note that Dylan and Java can prevent redefinition of certain classes. But, again, these are mechanisms which where not meant to be used for security. Using them to that purpose puts a strain on the design of class hierarchies.

Denial of service

Denial of service attacks can be mounted by consuming inordinate amounts of a shared finite resource. Typically either memory or processing time. Currently many executable content languages fail to provide any protection against those attacks [21][79]. The proper defence is to set up an accounting system for all shared resources and to impose strict quotas on each agent. The object-level ownership rules of TELESCRIPT make such accounting possible [82]. It is harder to prevent abuses of the communication subsystem [27].

---

1. It is not always straightforward to identify which classes are being exchanged between agents. So, to be on the safe side, all classes must have security code.

1.4 Summary

Message passing is a communication mechanism. Trying to use it to enforce security boundaries is a mistake. This is true even in a strongly typed object-oriented language with encapsulation. Encapsulation ensures that objects are accessed only via their interface, but these interfaces are not at the right level for security. First, interfaces are static, while access rights and trust are highly dynamic properties, i.e. an entity may gain or loose trust during execution but an object may not change it's interface. Secondly, interfaces do not differentiate between clients, all clients see the same interface. Security requires different levels of trust and thus would suggest multiple fine grained interfaces. Finally, checking security at the level of the object's interface implies that all intra-agent communication incurs the run-time cost, even if no inter-agent communication is used in the application. We believe that overloading message passing is confusing, inefficient and will lead to unnatural class design as security code will be spread all over the class hierarchy. An explicit agent-level communication mechanism, orthogonal with message passing, seems a more attractive solution.

2 Secure Object Spaces

We propose to investigate generative communication as an alternative to message passing for agent communication. The advantages of introducing a new concept are that agent communication is explicit and orthogonal to the other concepts of the language allowing the security effort to be better focused on uses of the new mechanism. Message passing retains its semantics and does not incur the extra cost of security.

The generative communication model of LINDA is well suited for coordination of parallel processes [15]. In LINDA, processes communicate by generating new data objects, called *tuples*, and writing them in a shared data structure, called the *tuple space*. This *tuple space* is an associative memory from which a process can retrieve tuples by pattern matching. Processes are viewed as active tuples which do not take part in pattern matching and turn into passive tuples when they terminate. LINDA was conceived to coordinate cooperating processes which had been designed together to solve one particular problem. We propose a new mechanism called *Secure Object Spaces* which enhances the LINDA model with security and accounting features and shift the emphasis from coordination to communication of independently developed processes with potentially conflicting goals.

2.1 Execution Model

A *Secure Object Space* (SOS) is data structure containing an active area, the *agent space*, in which a set of agents execute concurrently and a passive area, the *object space*, populated by tuples of objects. The SOS schedules the execution of agents in its active area, responds to memory allocation requests, and controls the reading and writing of tuples in its passive area. The SOS is also responsible for creating the initial environment of new agents and of granting them access to resources of the host. Note that *SOSs are not distributed*. Thus all tuple space communication is purely local to a single host. This is an important distinction as some LINDA systems offer distributed tuple spaces. Each agent is represented by the 5-tuple (a, $g^*(a)$, ID, PK, SK, T) where $g^*(a)$ is an object graph rooted at an object a, ID is the agent identifier (which is unique in the object space), PK is a public key, SK is a secret key, and T a set of threads of control. The passive area contains tuples of objects of varying length. The values stored in the tuples are restricted to a small set of built-in types, Integer, ImmutableString, Character, Real, PublicKey, or Capsule. All of these types are sealed to prevent being redefined in different agents. Most of these types are self explanatory, with exception of the Capsule type which will be discussed below. The core interface that the secure object space present to agents consists of five messages: out to output values, in and rd to retrieve values, mkKey to create a new private/public key pair and keyFor to get an agents key.

Pattern matching

To retrieve values from the object space, a simple form of pattern matching is used. Pattern matching relies on trying to match *tuples* with *anti-tuples*. An anti-tuple is a tuple which does not have all of its elements bound to values (*i.e.* a tuple with some "holes"). An element is either an actual or a formal. An actual maybe a Literal or a previously declared identifier of either Integer, ImmutableString, Character, Real, or PrivateKey type. A formal is a fresh identifier optionally followed by a type specification. Formals may not be of either PrivateKey or PublicKey types. A formal may also be the don't care symbol, written "_", which always matches. For an anti-tuple to match a tuple, all actuals of the anti-tuple must be equal to corresponding elements of the tuple, All formal must have type which are super types of the corresponding elements in the tuple.

Querying the SOS

To query the object space, the agent must create a block which consists of a set of anti-tuples. Each anti-tuple is associated with a list of statements. The semantics of the query operations (in and rd) are the following: (1) try to match the anti-tuple listed in the block, in turn, (2) if a match is found, bind the formals of the anti-tuple to the actuals of the tuple and execute the associated statement list, (3) otherwise block, until a matching tuple is written to the object space. SOS operations are atomic. The catch all pattern never blocks but does not bind to any value nor cause any tuple to be removed from the object space.

2.2 Public and private keys

Keys are an extension of the original LINDA model. They allow agents to use the object space for private communication. The principle is very simple, every tuple which contains a field with a PublicKey can only be matched by an anti-tuple containing the corresponding PrivateKey. New public/private key pairs can be created and it is possible for agents to communicate private keys. These keys are objects managed by the object space. Cryptography is not used here. The object space keeps track of the private keys known by each agent. Thus, it is not possible to guess somebody else's key. Keys are used to establish private communication between agents. In order to be able to hold a private exchange of more than a single message, agents may define a shared secret key. Note that it is not possible for a third party to know whether or not two agents are communicating.

2.3 Resource accounting

Accounting is under the care of the object space, which not only schedules the execution of agents but also keeps track of memory consumption. Charging agents for processing time is straightforward if tuples are "passive" while in the object space[1]. The memory used by each agent is equal to the size of the object graph of that agent, to which are added all tuples emitted by the agent and currently still in the object space. The ownership of a tuple changes when it is input by another agent (note that rd creates a copy of the matching tuple leaving the original tuple intact, and does not change ownership of that tuple). Issues such as expiration policies for old tuples and tuple garbage collection are currently being investigated.

3 Capsules

A capsule captures a portion of the state of an object system with the guarantee that no reference exist between the capsule's contents and the rest of the system. Thus a capsule contains an object graph that is disconnected from the rest of the system. The role of capsule in agent communication is crucial, as they represent the only way to exchange complex data structures between agents. If an agent wants to write a tuple containing arbitrary objects, it must first create a capsule with these objects and then write the capsule to the object space.

To create a capsule, it is necessary to identify, a portion of the object graph of the application, and to unlink it from the application so that no reference remain from inside the capsule to the outside and vice versa. A capsule is specified by a root object o and a list of fence post objects *Fence*. The capsule is subgraph of $g^*(o)$ containing only objects which can be reached from o without passing through any fence post $f \in$ *Fence*. Object in the set of fence posts *Fence* will be replaced by *placeholders*. Placeholders are abstract specifications of the object they replace. They are tuples: the first field is a type specification, the remaining are optional and encode additional information required to recreate the original object. Once a capsule has been constructed it is not possible to send message to the objects it contains as those objects are partially

1. In [46],[59] the pattern matching code is in the objects. This raises one question: who will be charged for the cycles spent pattern matching and how to prevent malicious match code.

unlinked. In that sense, the contents of the capsule are passive. To use a capsule's contents, it is necessary to open the capsule and provide replacement objects for all placeholders. A capsule can be opened only once.

If *o'* is the object returned by opening a capsule created at *o*, then $g^{*}(o)$ and $g^{*}(o')$ are disjoint. In other words, it is possible to give an arbitrary graph of object packaged in a capsule to an other agent, without any fear that this may open a security hole by which the agent may gain access to the sender. Similarly, the contents of a capsule can be used without fear that the sender has kept references on some of its parts and uses them to gain access to the receivers internals.

A capsule can also contain behaviours—the code of some of its objects. By default the behaviour is not included as it is assumed that it can be found in the receiving environment. Before opening a capsule an agent may check whether it contains any behaviour, if it does then the agent must decide whether it is safe to open the capsule, as that behaviour will be injected in the current environment.

Capsule can capture as much the state of an agent as needed, including the state of all threads of control and all the attached code, or as little as a single object. The usefulness of capsules for communication comes from the fact that they create disjoint subgraphs from the main object system. Capsules are also used for mobility to package the system state. For the sake of mobility it is crucial to control very tightly the amount of data transferred. Placeholders are thus used to limit the size of the object graph to store in the capsule, and they also define the point where to reconnect the contents of the capsule to the environment.

A trivial way to create a capsule is to copy the object graph and transmit the copy as is done with marshalling code [10]. The amount of copying can be reduced by adapting the proposal of Hogg [39]. These issues are currently being studied. Capsules can also be implemented in some pickle-based systems [14][10].

4 Conclusion

This working paper is a first attempt at providing secure communication mechanisms for a mobile object systems. We introduced an new communication mechanism, orthogonal to message passing, specifically designed for agent communication which guarantees security of agent communication by keeping the agents disjoint. As this is still work in progress, we have little evidence to support our claim that Secure Object Spaces are appropriate for secure communication in mobile object systems. Experience with real programs is needed for that.