

Secure Mobile Agents – SeMoA

Volker Roth, Mehrdad Jalali, Ulrich Pinsdorf

German Abstract

Mobile Agenten sind Softwareprogramme, die von einem Rechner im Internet zum anderen »springen« können und dabei autonom nach Informationen suchen oder anderen Aufgaben nachgehen. Mit mobilen Agenten werden verteilte Systeme an ihre Grenzen geführt, denn nicht nur die Berechnungen können verteilt werden, sondern auch der Programmcode mit dem diese Berechnungen durchgeführt werden. Mit Hilfe von mobile Agenten Technologie können Algorithmen zur Filterung von Daten direkt zur Quelle der Daten transportiert werden und nur die Ergebnisse der Filterung müssen anschließend über das Netzwerk übertragen werden. Da mobile Agenten ihren Programmcode mit sich führen, kann die Filterung auch im disconnected mode erfolgen, d.h. der Rechner von dem aus ein mobiler Agent versendet wird muß keine permanente Netzverbindung aufrecht erhalten. Die Netzverbindung ist erst wieder erforderlich wenn der mobile Agent zurückkehren oder seine Ergebnisse übertragen möchte. Um solche und weitere Vorteile dieser Technologie nutzen zu können, muß die verwendete Mobile Agenten Technologie mit einer Reihe von Bedrohungen fertig werden können. Die Route eines mobilen Agenten kann mehrere Server beinhalten, auch solche von konkurrierenden Betreibern. Ein bösartiger Server könnte die Daten eines Agenten verändern, einsehen, oder aber so modifizieren, daß der Agent auf anderen Servern Schaden anrichtet. Andererseits könnte ein Agent auch versuchen den Server anzugreifen oder andere Agenten, die von ihm beherbergt werden.

Introduction

Mobile agents push the flexibility of distributed systems to their limits since not only computations are distributed dynamically, but the code that performs them is also distributed. Once send out, mobile agents roam the network, seek information, and carry out tasks on behalf of their senders autonomously. Upon return, the agents present all results to their senders. Meanwhile, the user does not need to monitor the application's progress permanently. This makes mobile agents particularly useful in mobile environments (disconnected operations), because no permanent network connection is necessary to run an agent-based application. Mobile agents also offer great benefits to applications in »wired« networks by adding client-side intelligence and functionality to server-side services unified under a homogenous access paradigm. Furthermore, mobile agents offer considerable network bandwidth savings because they can migrate to, and process data from the source of that data, which therefore do not need to be send back and forth across the network.

In order to exploit benefits such as the ones described above, mobile agent frameworks have to cope with a number of security threats. A mobile agent's itinerary usually spans a number of servers which might be run by competing operators. Apart from monitoring, manipulating, and stealing data from mobile agents, *malicious hosts* might try to abuse passing agents as Trojan Horses in attacks on competing servers while incriminating the agent's owner in the process. On the other hand, hosts have to be aware of *malicious agents* breaking into their

server harming other hosted agents, or to gaining unauthorized system access. Mobile agents make a perfect cover for viruses and Trojan Horses. In particular, mobile agents might try to attack remote hosts while using innocent hosts as launch pads in order to cover the tracks of the attacker. Both agents and servers are threatened by attacks from outside the system. Eavesdroppers might snoop on agents which are transferred over network connections and hence compromise the privacy of agents. They might also launch active attacks on servers either directly or indirectly by manipulating agents during transport. A sound security model which is able to resist these attacks is fundamental to business acceptance and market exploitation of this fascinating technology.

Security Architecture

SeMoA bases on JDK 1.3 and is an effort to provide adequate security for mobile agent systems, servers as well as agents. The security architecture of a SeMoA server can be compared to an onion: agents will have to pass all layers of protection before they are admitted to the runtime system (cf. Figure 1 for illustration) and the first class of an agent is loaded into the server's JVM.

The first (outer) security layer is a transport layer security protocol such as TLS or SSL. This layer provides mutual authentication of agent servers, transparent encryption and integrity protection. Connection requests of authenticated peers can be accepted or rejected as specified in a configurable policy.

The second layer consists of a pipeline of *security filters*. Separate

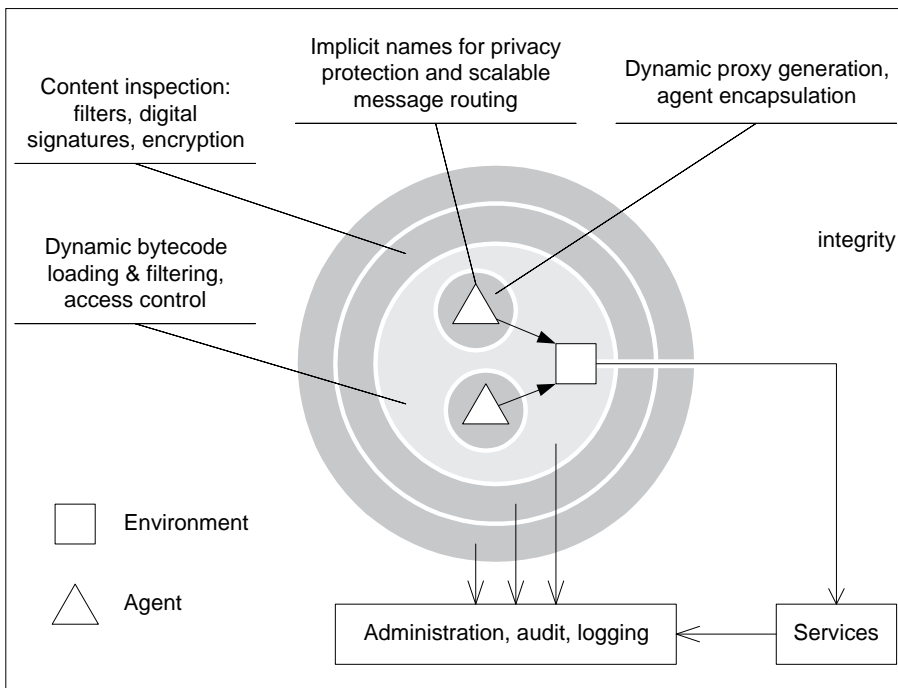


Figure 1: SeMoA's security architecture may be compared to an onion – incoming agents have to pass all of several layers of security functions before they are admitted to the system

pipelines for incoming agents and outgoing agents are supported. Each filter inspects and processes incoming/outgoing agents, and either accepts or rejects them. We also refer to this filtering procedure as *content inspection*, in analogy to concepts known from *firewalls*. At the time of writing, SeMoA features two complementary pairs of filters that handle digital signatures and the selective encryption of agents (signature verification and decryption of incoming agents, encryption and signing of outgoing agents). An additional filter at the end of the incoming pipeline assigns a configurable set of permissions to incoming agents, based on information established and verified by preceding filters. Permissions can be granted based on the authenticated identities of the agent's owner, the sponsor of its last state change, and its most recent sender. Filters can be registered and unregistered either dynamically or at boot time of the SeMoA server either programmatically or by means of script executed at boot time.

Subsequent to passing all security filters, SeMoA sets up a sandbox for the accepted agent (which can

be regarded as layer four). Each agent gets a separate thread group and class loader. The agent is unmarshaled by a thread that is already running within the agent's thread group and becomes the first thread of that agent. Marshalling is done by the very same thread after all remaining threads in the agent's thread group have terminated. As the Serialization Framework of Java provides callbacks that pass the control back to objects to be serialized, the thread once again blocks until no more threads remain in the agent's thread group. Only then does SeMoA handle any migration requests of that agent. This prevents agents from flooding a network of agent servers by migrating and refusing to terminate at the same time.

An agent's classes are loaded by its defined class loader. This class loader supports loading classes that came bundled with the agent, as well as loading classes from remote code sources specified in the agent. All loaded classes (except those in the class path of the server) are verified against a configurable set of trusted hash functions. The digests of verified

classes must match corresponding digests signed by the agent's owner (which can be regarded as layer three). Thus, only classes authorized by the agent's owner for use with his agent are loaded into the agent's namespace.

As agents cannot share classes, one agent cannot load a Trojan Horse class into the name space of another agent. However, in order to allow method invocations between agents, they may share interfaces. Interfaces are deemed to be the same if their trusted digests match. In this case, an agent's class loader returns a previously loaded interface rather than loading the interface again from the served agent, so that the interfaces used by the agents are type-compatible wherever possible.

Before a class is defined in the Java VM, a bytecode of that class has to pass a filter pipeline similar to the one for incoming agents. Each class filter can inspect, reject, and even modify the bytecode. SeMoA is equipped with an example filter that rejects classes which implement `finalize`. Malicious agents may implement this method in order to attack the garbage collector thread of the hosting VM. Additional filters may implement bytecode arbitration, e.g. in order to add resource accounting to agent classes.

Agents are separated from all other agents in the system, no references to agent instances are published by default. The only means to share instances between agents is to publish them in a global *environment*. Each agent gets its own view on this environment (referred to as the agent's environment), which tracks the objects registered by that agent. All published objects are wrapped into proxys which are created dynamically. If the agent terminates or retracts a published object, then the *agent's environment* instructs the handler of the corresponding proxy to invalidate its link to the original object. This makes the original object unavailable even to other agents that looked up its reference in the

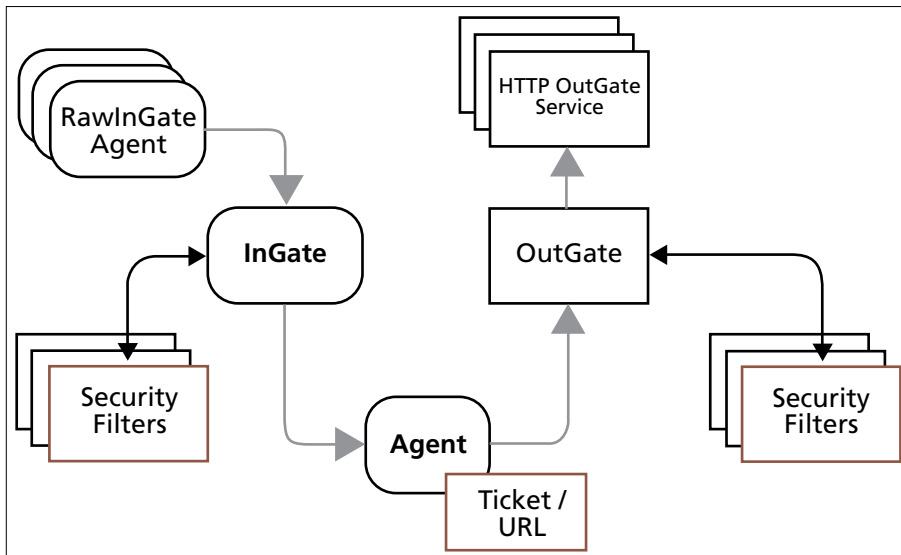


Figure 2: One layer of protection consists of pipelines of security filters. Filters operate on incoming and outgoing agents, and they can reject agents if they violate the server's security policy (e.g. if virus signatures are detected in an agent's code)

global environment. Moreover, it makes the original object available for garbage collection.

Agent Structure

In SeMoA, mobile agents are transported as Java Archives (JAR files). The JAR specification of Sun Microsystems extends ZIP archives with support for digital signatures by means of adding appropriate signature files to the contents of the ZIP archive. The signature format is PKCS #7, a cryptographic message syntax standard which builds on standards such as ASN.1, X.501, and X.509. Using PKCS #7 as well, SeMoA extends the JAR format with support for selective encryption of JAR contents with multiple recipients. Encryption and decryption is handled transparently for agents by the filters described in the previous section. In order to prevent encrypted parts of an agent from being copied and used in conjunction with other agents (*cut & paste* attacks), these filters implement a non-interactive proof of knowledge of the required decryption keys.

Each agent bears two digital signatures. The entity that signs the *static* part of an agent (the part that remains unchanged

throughout the agent's lifetime) is taken as the rightful *owner* of that agent (the entity on whose behalf the agent is acting). Each sending server also signs the complete agent (static part plus *mutable* part) – therefore it binds the new state of the agent to its static part. In other words, agent servers commit to the state changes that occurred to an agent while they hosted this agent.

In addition, SeMoA computes implicit names from agents, by applying the SHA1 digest algorithm to its owner's signature. This renders unique as well as anonymous agent names globally. *Implicit names* are used in SeMoA to provide agent tracing as well as scalable location-independent routing of messages among agents.

Other Features

One of the most striking features of the SeMoA architecture is its openness, both in terms of adding new features and supporting interoperability among different agent systems. Agents are not managed by the server directly but by using an abstraction called *Lifecycle*. A *Lifecycle* is the system's view on the agent, wraps around a matching agent, provides a

familiar view of the system to the agent, and translates between the native SeMoA view and the one which is expected by the agent. Different Lifecycle factories can be registered in the SeMoA server, thus multiple agent systems can be emulated at the same time, potentially allowing heterogenous agent types to interoperate. However, the quality of the emulation depends on the quality of the design of the emulated system. The clearer and the more concise the interfaces of a system are, the easier can it be integrated and emulated. As a consequence, no special class needs to be the subclass of agents in order to run in the SeMoA server. The server is equipped with a default *Lifecycle* which supports the *Runnable* interface, the easiest way to start the Java code. SeMoA has been deployed successfully in the ESPRIT Project AIMedia, and is also used in the MAP (Multimedia Arbeitsplatz der Zukunft) project which is supported by the BMWi (Bundesministerium für Wirtschaft).

Points of contact

Volker Roth
 Mehrdad Jalali-Sohi
 Fraunhofer IGD, Darmstadt,
 Germany
 Email: vroth@igd.fhg.de
 jalali@igd.fhg.de