



Secure Mobile Agents

Ulrich Pinsdorf

German Abstract

Mobile Agenten sind Software-Programme, die von Rechner zu Rechner »springen« können und dabei autonom nach Informationen suchen. Mit mobilen Agenten werden verteilte Systeme an ihre Grenzen geführt, denn nicht nur die Berechnungen können verteilt werden, sondern auch der Programmcode. Mit Hilfe von mobilen Agenten können z.B. Suchalgorithmen direkt zur Quelle der Daten transportiert werden. Lediglich die Suchergebnisse müssen anschließend über das Netzwerk übertragen werden. Um diese Technologie erfolgreich nutzen zu können, muss eine Infrastruktur für mobile Agenten mit einer Reihe von Bedrohungen umgehen können. Einerseits muss der Agent vor ungewünschter Veränderung, Ausspähung und Modifikation durch bössartige Server geschützt werden. Andererseits müssen Server und andere Agenten vor potentiell bössartigen Agenten geschützt werden. Im Rahmen des Projektes SeMoA wird eine Infrastruktur für mobile Agenten entwickelt, der sich dieser Sicherheitsanforderungen annimmt. Dabei ist die Interoperabilität von Agentenplattformen ein wichtiger Aspekt. So ermöglicht SeMoA das Ausführen von Agenten, die eigentlich für andere Systeme entwickelt wurden. SeMoA bietet eine Vielzahl von Sicherheitsmechanismen auf Basis weit verbreiteter Industrie- und Internetstandards, ist komplett in Java entwickelt und als Open Source frei verfügbar.

SeMoA (Secure Mobile Agents) is an initiative of the Fraunhofer Institute for Computer Graphics, Department Security Technology for Graphics and Communication Systems, to create a platform for secure multimedia and electronic commerce applications based on mobile agent technology. Keywords: mobile agents, security, interoperability, mobile computing.

Introduction

Mobile agents push the flexibility of distributed systems to their limits since not only computations are distributed dynamically but also the code that performs them. Once let loose, mobile agents roam the network, seek information, and carry out tasks on behalf of their senders autonomously. Upon return to their senders, they present the results of their endeavors. These mobile agents have the positive effect of freeing the user from permanently monitoring the application's progress which makes mobile agents particularly useful in mobile environments (disconnected operation), because no permanent

network connection must be maintained in order to run the 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 directly at the source of that data, which therefore does not need to be shipped back and forth across the network.

In order to exploit benefits (as described above), mobile agent frameworks have to cope with a number of security threats. A mobile agent's itinerary in general 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.

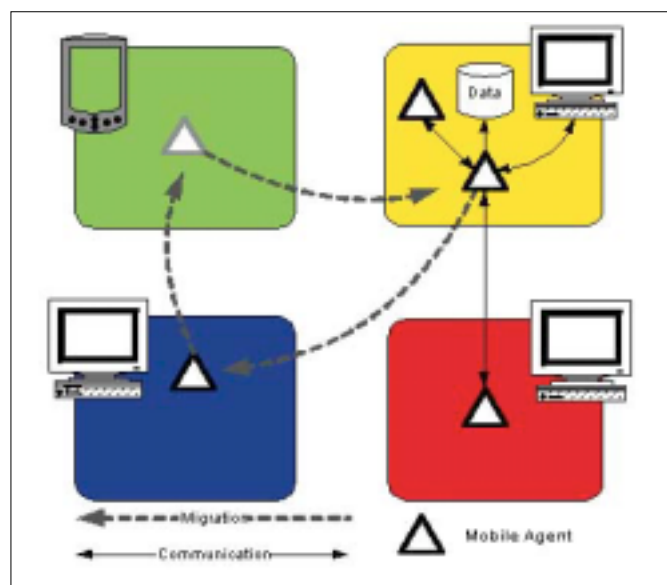


Figure 1: Mobile agents are software entities that migrate back and forth in a network. Furthermore they may access data storages, and communicate with other agents in order to fulfill a specific task

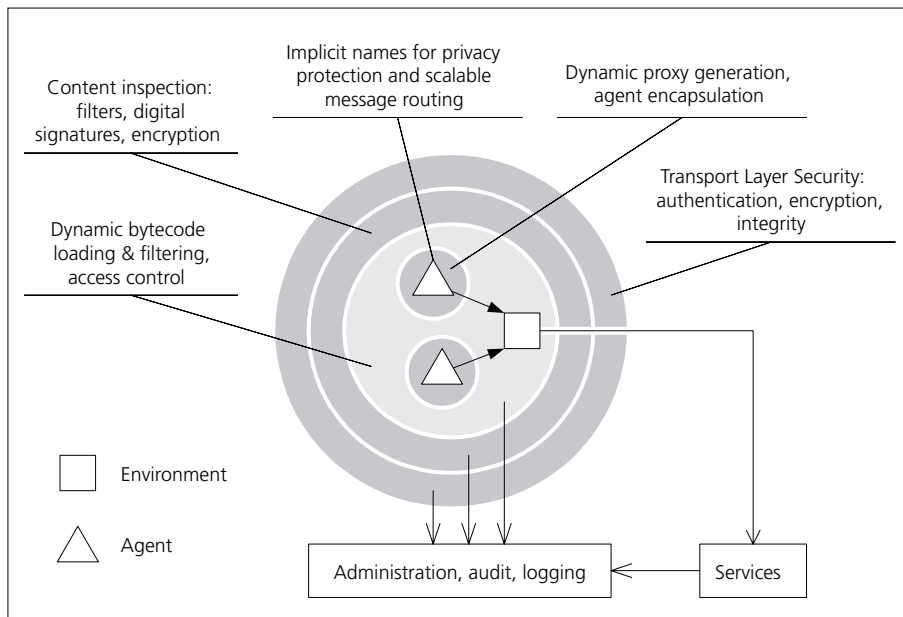


Figure 2: The general security architecture is comparable to an onion; agents have to pass all of several layers in order to be admitted to the runtime system

On the other hand, hosts have to be aware of *malicious agents* breaking into the server in order to harm other agents hosted by them, or to gain unauthorized system access. Mobile agents make a perfect cover for viruses and Trojan Horses. In particular, mobile agents might try to attack remote hosts using innocent hosts as launch pads to cover the tracks of the attacker. Both agents and servers are threatened by attacks from outside the system. Eavesdroppers might snoop on agents that are being transferred over network connections hence compromising 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 builds on Java technology and is a »best effort« to provide adequate security for mobile agent systems, servers as well as agents. The security architecture of the SeMoA server can be compared with an onion: agents have to pass all layers of protection before they are admitted to the runtime system (see 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 pipelines for incoming agents and outgoing agents are supported. Each filter inspects and processes incoming/outgoing agents, and either accepts or rejects them. We refer to this filtering procedure as *content inspection* in analogy to concepts known from Internet *firewalls*. SeMoA features two complementary pairs of filters that handle digital signatures and 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 a certain security policy. This policy implements a role-based access control mechanism that is mandatory for each agent. Permissions can be granted based on the authenticated identi-

ties of the agent's owner, the sponsor of its last state change, its most recent sender, the name of the Java class, and many more. Filters can be registered and unregistered either dynamically or at boot time of the SeMoA server (either programatically or by means of script executed at boot time).

Subsequent to passing all security filters, SeMoA sets up a sandbox for the accepted agent (layer four). Each agent gets a separate thread group and class loader. The agent is unmarshalled 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. Since 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 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.

Agent's classes are loaded by its dedicated class loader. This class loader supports loading classes that came bundled with the agent, and loads classes from remote code sources specified by the agent. All loaded classes (save 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 (layer three). Thus, only classes authorized by the agent's owner for use with his agent are loaded into the agent's namespace. Agents cannot share classes, i.e. 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 will 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, the 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. This is comparable with protection mechanisms used by virus scanners. 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*, which represents a hierarchical name space for objects. 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 its 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 global environment. This also 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 agent names globally unique as well as anonymous. Implicit names are used in SeMoA to provide agent tracing as well as scalable location-independent routing of messages among agents.

Interoperability

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, which 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 expected by the agent. Different lifecycle factories can be registered on the SeMoA server, thus multiple agent systems can be emulated in parallel, allowing heterogenous agent types to interoperate. Currently, SeMoA supports lifecycles for the well-known mobile agent platforms JADE, Tracy and Aglets which means that agents initially programmed for these platforms may run within a SeMoA server without any modification. They participate from all SeMoA features mentioned above, and benefit in particular from the secure run-time environment. SeMoA has been deployed successfully in the ESPRIT Project AIMedia, in the projects MAP (Multimedia Arbeitsplatz der Zukunft) and MOBILE which are both supported

by the BMWi (Bundesministerium für Wirtschaft) and BMBF (Bundesministerium für Bildung und Forschung).

Further information

<http://www.semoa.org>

Points of contact

Ulrich Pinsdorf
Jan Peters
Peter Ebinger
Fraunhofer IGD, Darmstadt,
Germany
Email:
ulrich.pinsdorf@igd.fraunhofer.de
jan.peters@igd.fraunhofer.de
peter.ebinger@igd.fraunhofer.de