Concepts and Architecture of a Security-Centric Mobile Agent Server

Volker Roth   Mehrdad Jalali-Sohi

Fraunhofer Institut für Graphische Datenverarbeitung
Rundeturmstraße 6, 64283 Darmstadt, Germany

E-mail: {vroth|jalali}@igd.fhg.de

Abstract

Mobile software agents are software components that are able to move in a network. They are often considered as an attractive technology in electronic commerce applications. Although security concerns prevail. In this paper we describe the architecture and concepts of the SeMoA server – a runtime environment for Java-based mobile agents. Its architecture has a focus on security and easy extendability, and offers a framework for transparent content inspection *of agents by means of* filters. *We implemented filters that handle agent signing and authentication as well as selective encryption of agent contents. Filters are applied transparently such that agents need not be aware of the security services provided by the server.*

Keywords: *autonomous agents, e-commerce, mobile agent security, malicious host, distributed applications*

1 Introduction

Mobile agents [1] push the flexibility of distributed systems to their limits since not only computations are distributed dynamically, the code that performs them is also distributed. Information gathering and electronic commerce are application areas in which mobile agent technology may offer substantial benefits [2]. In the course of the European ESPRIT Project AIMedia (Targeted Advertising on Interactive Media) we developed a proof of concept application based on mobile agent technology by which we demonstrated some of the benefits that can be expected from using mobile agent technology (e.g. delegating a high-level shopping task to a mobile agent [3]). In this article, we describe the basic concepts and architecture of the mobile agent server that was the basis of said application. The server is still under active development at the Fraunhofer Institute for Computer Graphics.

Security has been identified numerous times by different researchers as a top criterion for the acceptance of mobile agent technology. While some researchers decided to concentrate on mobile agents in general, others (including us) decided to investigate the security aspects in particular. The title of our platform – *Secure Mobile Agents* (SeMoA) – was chosen to reflect this focus. We discuss the security mechanisms supported by SeMoA in Sections 3 to 7. In Section 9 we summarize some open problems. Section 8 highlights some related work and conclusions are drawn in Section 10.

2 Mobile Agents Benefits

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 the agents present the results of their endeavors. Meanwhile the user is freed of the obligation to permanently monitor the application's progress. This 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, at the source of that data, which therefor need not be shipped back and forth across the network. Applications

Copyright 2002 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

based on mobile agents are inherently distributed. Agents are often independent of a particular hardware or operating system, and can be deployed in heterogenous environments.

Several further advantages were claimed for mobile agent in addition to those summarized above [2]. A recommendable overview of some of the more frequently quoted and generally accepted claims is given in [4].

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 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. On the other hand, hosts have to be aware of *malicious agents* breaking into the server in order to harm other agents hosted by it, or to gain unauthorized system access. Mobile agents make a perfect cover for viruses, worms, and Trojan Horses. In particular, mobile agents might try to attack remote hosts using innocent hosts as launch pads in order to cover the tracks of the attacker. Both agents and servers are threatened by attacks originating from outside the system. Eavesdroppers might snoop on agents being transfered 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. Network-based attacks on mobile agent systems do not add new threats when compared to client/server applications, although the impact of a succesfull attack might be much worse. A sound security model which is able to resist these attacks is fundamental to business acceptance and market exploitation of this fascinating technogy.

3 General Architecture

On designing the SeMoA server we decided that it shall consist of a minimum lightweight core that is easily extended by means of (static) agents and a plugin mechanism. This facilitates tailoring a server core to the specific needs of a particular application domain. We did not want to fix a set of security mechanisms beforehand but to keep the security mechanisms extendable and adaptable to the requirements. Any cryptographic protection mechanisms used in SeMoA are designed and implemented on top of the JCA/JCE (Java Cryptography Architecture & Extension [5, 6]) in order not to become dependent on a single cryptographic technology. The tasks of the SeMoA server can be summarized as follows:

- Load and install agents in the system.
- Sending stop signals to agents.
- Remove and dispatch agents.
- Maintain separation and anonymity of agents.
- Provide service management.

The architecture we came up with distinguishes between *agents* and *services*. Agents own resources such as threads and persistent storage space while services are meant to be called by agents and hence run in the caller's thread. Agents may register, unregister, and request services; such services can refer to (and thus borrow resources from) their owners.

Agents are maintained in a map which is indexed by agent name. The agents map is a private object of the server object. The value of a mapping defined in the agents map is the *agent context*. The agent context does *not* represent the agent's environment (as for instance the `AppletContext` does with respect to an Applet) but is an object that controls the lifecycle of the agent it represents, and maintains references to the various objects associated with an agent. This wording is due to historical reasons and differs from the one used by some other agent systems such as *Aglets*.

Service management is handled by the service *registry* which is described in greater detail in Section 5. It is used throughout the system as a shared space for publishing objects.

4 Security Architecture

SeMoA builds on JDK 1.3[1] 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 compares to an onion: agents have to pass all of several layers of protection before they are admitted to the runtime system (see Figure 2 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 [7] or SSL [8]. At the time of writing, the SSL implementation is used that comes with the *Java Secure Socket Extension* (JSSE) framework provided by Sun Microsystems. 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 also as *content inspection* in analogy to concepts known from firewalls. At the time

[1] We recently moved from JDK 1.2 to JDK 1.3 to make use of its novel proxy generation features.

of writing, 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 information established and verified by preceeding 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 programatically or by means of configuration files.

Subsequent to passing all security filters, a sandbox is set up for the accepted agent (which can be regarded as 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 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.

The classes brought by an agent are annotated with *tag permissions* which are generated dynamically and which are unique for each agent. Whenever e.g. an agent attempts to modify a thread group the current thread is traced back to the corresponding agent's thread group which then identifies a tag permission to test. This prevents agents from manipulating threads of other agents even if one agent invokes methods of another. A configurable *threads filter* sorts threads created by "special" classes (e.g. classes of the *Abstract Window Toolkit* (AWT)) into separate thread groups so that these do not interfere with agent threads.

An agent's classes are loaded by its dedicated class loader. This class loader supports loading classes that came bundled with the agent, as well as loading classes from remoted code sources specified in 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 (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.

Agents cannot share classes so one agent cannot not load a Trojan Horse class into the name space of any other 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. Of course, this works only if the interface classes referenced by two agents are bitwise identical. Improved schemes may compare interface implementations on the API level. However, this adds overhead to the class loading process, and is not yet implemented. Interface objects are managed in a `Map` that is global to all agent class loaders. Pollution attacks on this `Map` require breaking the trusted hash functions.

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. SeMoA comes 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 the *registry*. Each agent gets its own view on the registry (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 registry. Furthermore this makes the original object available for garbage collection.

5 The Registry

The registry maintains any number of service *levels* subject to configuration in the server's configuration file. Each level is identified by a unique name, and for each level the registry allows to:

- Publish objects under a given name.
- Retract by name.
- Lookup objects by name.
- List the object names.

Furthermore it allows to list the names of all known levels. All operations in the given list are subject to a permission check. Permissions can be granted based on the level name, service name, and operation. The asterisk can be used in permissions as a wildcard character for all levels

or all names. The server does not allow asterisks in ordinary names in order to prevent clashes with wildcards.

Service objects published by SeMoA store the access control context [9] which is current upon creation in a private variable. Privileged actions of these services set the stored access control context. This prevents leakage of privileges due to the global visibility of classes in the local class path. Though malicious code might instantiate service objects directly without consultation of the registry this does not give the code extended privileges because the permissions of the freshly created instance are limited to those granted to the access control context of the malicious code. In other words agents are required to retrieve the service instance maintained by the registry which is subject to access control.

The registry is meant to be the low-level bootstrapping mechanism for agents and services to find out about other services. Any high-level services such as facilitators of agent communication languages are made available by means of the registry. Agents must be addressed by means of services, never directly.

6 Agent Transport and Content Inspection

The basic configuration of a SeMoA server registers two *portal* services for agent transportation on the *transport* level. The first one, called *ingate*, is the portal into the server. The second, called *outgate*, is the portal out of the server (see Figure 3).

When an agent passes through a portal then the portal first scans the level *security* for *filter* services (either of the *incoming* or *outgoing* type). The agent (more precisely: its context) is then handed to each filter in turn for inspection and annotation. Filters are arranged in the lexical order of their names. Each filter either accepts or rejects the agent. At the time of writing, the portals simply dispose of rejected agents.

Neither portal handles transportation on its own. The *ingate* is served by transport services that e. g. listen on network ports. Transport services simply pass a received agent to the *ingate* which decompresses it and takes care of further processing. The *outgate* is triggered by an event that is fired when an agent terminates. This event indicates the name of the agent in question. The *outgate* then filters the agent as described above. If the agent set a *ticket* then the *outgate* scans the *transport* level for a transport service that accepts the ticket. The first transport service that agrees in transporting the agent wins. A ticket consists a set of alternative URLs pointing at the desired destination. See Figure 1 for illustration.

The transport services enforce a (configurable) maximum size on the decompressed size of agents. If the size exceeds the allowed maximum then the decompression is aborted and the agent is discarded. SeMoA comes with services that support agent transport via simple socket connections (either plain or SSL sockets) and HTTP transport. HTTP transport is handled by means of a Web agent and a Servlet registered with it. Details of how the Web is integrated in SeMoA can be found in [3].

7 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 [10], 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 introduced in Section 3. 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 (see [11] for details).

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); therefor 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. An early discussion of these concepts can be found in [12].

Agents can use abstractions comparable to the combination of a `Map` interface and a filesystem in order to access and store data in their static and mutable part (denoted its *structure*). For instance, a similar *folder* abstraction was used already in TACOMA [13]. When an agent migrates, its structure is processed by the outgoing security filters, and is compressed back into a JAR for transport to its destination host. The marshalled instance graph of an agent is also stored in the agent's structure. As a side effect, agents can schedule the computation of snapshots as they see fit. Agent structures can be backed both by persistent and non-persistent storage, depending on the server's configuration. The agent structure also contains properties of the agent (key/value pairs), such as a human readable nickname of the agent and code sources to load classes from. The properties must be signed by the agent's owner, thus they are protected against tampering.

In addition, SeMoA computes *implicit names* [14] 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, and will be used for scalable location-independent routing of messages among agents as well.

In summary, SeMoA supports four types of access rights for the folders of an agent:

Read-only: This data can be read on each host but cannot be modified without breaking the agent's verifiable integrity.

Read/write committed: This data can be read and modified on each host but hosts have to commit to the new state. The changes can (in principle) be checked and linked to that host on the agent's next hop.

Group read: This data can be read only on a predetermined set of authorized hosts. Modification of the data breaks the agent's verifiable integrity.

Group read/write: This data can be read and modified only on a predetermined set of authorised hosts.

Groups of valid recipients can be defined flexibly. The data a mobile agent gathers on one host can be protected against eavesdropping by hosts not belonging to the access group of the folder in which the data is stored. Since the protection mechanisms are part of the server's security services, agents can remain unaware of the cryptographic operations and key management. The structure of an agent's JAR file is given below:

META-INF/	MANIFEST.MF	
	OWNER.SF	
	OWNER.(DSA|RSA)	
	SENDER.SF	
	SENDER.(DSA|RSA)	
SEAL-INF/	INSTALL.MF	
	name$_i$.EAR	$i = 1, \ldots, n$
	name$_j$.P7	$j = 1, \ldots, m$
static/	agent.properties	
mutable/	instance.ser	

The file `agent.properties` consists of name/value pairs. The properties as well as the initial classes brought by an agent are covered by the owner's signature, thus any modification of the properties breaks the static part's integrity. The file `instance.ser` contains the serialized object instance graph of the agent. The information in `SEAL-INF` is used to manage selective encryption of agent contents.

8 Related Work

The *Mobile Agent List* [15] gives an impression of the variety of current mobile agent systems. A number of systems on this list share with SeMoA a certain bias towards security issues, most notably Mole [16], D'Agents [17], Ajanta [18], and JavaSeal [19].

For instance JavaSeal is more rigorous in its separation of agents than our system. The price for the improved separation is paid in terms of reduced performance and a restriction of the Java environment that is available to the seals (agents). Seals communicate by means of synchronous channels; objects that are communicated are passed by value in order to prevent sharing of object references between seals. The JavaSeal packages almost completely replace the standard JDK packages. Sharing of classes between seals is minimized. While this improves the security against DoS attacks and covert channels it also means that identical copies of many classes must be loaded by the class loaders of the individual seals.

Ajanta provides a number of mechanisms which are comparable to the ones we deploy in SeMoA. Agents can have a *read-only* state which is protected by means of digital signatures. A *targeted state* is used to reveal parts of the agent's state to selected recipients. Each object in the targeted state is encrypted with the public key of its intended recipient. However, it is not clear how Ajanta prevents an adversary from cutting encrypted objects out of the targeted state, and pasting them into the targeted state of an agent of his own. All the adversary then has to do is to send his agent to the listed recipients where his agent asks the recipients to decrypt the objects. Then it brings back the plaintext objects to the adversary.

9 Open Problems

We decided to build SeMoA on a regular Java Virtual Machine (VM) with unmodified core packages. This rules out a number of protective measures that are, in principle, expected from a viable and secure mobile agent platform. The foremost shortcoming is the lack of proper resource control in the Java VM. As a consequence, SeMoA is not robust against a number of DoS (Denial-of-Service) attacks such as memory exhaustion.

Another problem is the forced termination of agents or – more precisely – the termination of threads. All methods that allow to stop threads are deprecated in Java 2 and using them anyway provokes inconsistent object states. Even if `stop()` is called on the thread of an agent then the agent might still catch the resulting `ThreadDeath` (or any other `Throwable`) that is propagated up the thread's stack and continue. For the time being we chose to ignore this problem. The SeMoA server flags down agents in case they

should terminate, and builds on their cooperation.

Last not least there are a number of classes in the Java core package which synchronize on the class object itself. Since local classes are shared and their visibililty is global any agent that acquires a lock on such a class object effectively blocks any other threads attempting to access them.

Some of these issues can be dealt with by means of dynamic byte code rewriting [20] as well as extended byte code analysis, restrictions on the visibility of core classes, and minimization of shared classes as described for instance by Bryce and Vitek [19]. We did not yet implement said protective mechanisms though the architecture of SeMoA supports easy integration of filters suitable for this purpose.

In summary, a number of shortcomings of Java can be used by malicious agents to launch various DoS attacks. However, in order to do so, agents need to run first. But before an agent is run (and even before classes of the agent are loaded), SeMoA servers verify the claimed identity of the agent's owner based on digital signatures and certificates. Though the culprit can try to cover his tracks, there is a chance that he can be tracked by means of evidence logged to a secure host.

10 Conclusions

In this article we gave an overview of the basic architecture of SeMoA, and we highlighted some of its features that set it apart from other systems. Most notably, SeMoA is open to a multitude of transport protocols and supports flexible content inspection of incoming and outgoing agents by means of filters. Adding new filters is simple. The server core can be easily tailored to the requirements of a particular domain by adding the appropriate services. Two complementary pairs of filters handle initial agent authentication and signing as well as confidentiality of selected folders in the agent's structure. Cut & paste attacks on the encrypted contents are detected and prevented. Furthermore, we introduced the notion of tag permissions which are generated dynamically and which are used to implement per agent access control to thread groups. The server architecture proved to be very flexible and powerful. We are constantly enhancing the server, among the next things we plan to do is to provide extended integrity protection of agents based on mechanisms described by Karjoth et al. [21], as well as algorithms based on co-operating agents [22].

11 Acknowledgements

Parts of this work were sponsored through the ESPRIT project *AIMedia: Targeted Advertising on Interactive Media*, project number 26983.

References

[1] J. E. White, "Mobile agents," in *Software Agents* (J. Bradshaw, ed.), ch. 18, pp. 437–472, Menlo Park, CA: AAAI/MIT Press, 1997.

[2] D. B. Lange and M. Oshima, "Seven good reasons for mobile agents," *Communications of the ACM*, vol. 42, pp. 88–89, March 1999.

[3] V. Roth, M. Jalali, R. Hartmann, and C. Roland, "An application of mobile agents as personal assistants in electronic commerce," in *Proc. 5th Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000)* (J. Bradshaw and G. Arnold, eds.), (Manchester, UK), pp. 121–132, April 2000. ISBN 1-902426-07-X.

[4] T. Papaioannou, *On the Structuring of Distributed Systems: The Argument of Mobility*. Ph.d. thesis, Loughborough University, February 2000.

[5] Sun Microsystems, Inc., *Java™ Cryptography Architecture API Specification & Reference*, July 1999. Internet document available at URL: `http://java.sun.com/j2se/sdk/1.3/docs/guide/security/CryptoSpec.html`.

[6] Sun Microsystems, Inc., *Java™ Cryptography Extension API Specification & Reference*, 1999. Not available outside the U.S.A.

[7] T. Dierks and C. Allen, "The TLS protocol version 1.0," Request for Comments 2246, Internet Engineering Task Force, jan 1999.

[8] A. Frier, P. Karlton, and P. Kocher, "The SSL 3.0 protocol." Netscape Communications Corp., nov 1996.

[9] L. Gong, *Java™ Security Architecture (JDK 1.2)*. Sun Microsystems, Inc. in [23], relative URL: `file:/docs/guide/security/spec/security-spec.doc.html`.

[10] RSA Laboratories, "Cryptographic message syntax standard," Public Key–Cryptography Standards 7, RSA Laboratories, Redwood City, CA, USA, 1993. Available at URL: `ftp://ftp.rsa.com/pub/pkcs/`.

[11] V. Roth and V. Conan, "Encrypting Java Archives and its application to mobile agent security," in *Agent Mediated Electronic Commerce: A European Perspective* (F. Dignum and C. Sierra, eds.), vol. 1991 of *Lecture Notes in Artifical Intelligence*, pp. 232–244, Berlin: Springer Verlag, 2001.

[12] V. Roth and M. Jalali, “Access control and key management for mobile agents,” *Computers & Graphics, Special Issue on Data Security in Image Communication and Networks*, vol. 22, no. 4, pp. 457–461, 1998.

[13] D. Johansen, “Mobile agent applicability,” in Rothermel and Hohl [24], pp. 80–98.

[14] V. Roth, “Scalable and secure global name services for mobile agents.” 6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages (Cannes, France, June 2000).

[15] F. Hohl, “The mobile agent list.” Internet document available at URL `http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.ht%ml`. Version current 11 October 1999.

[16] M. Straßer, J. Baumann, and F. Hohl, “Mole: A Java based mobile agent system,” in *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems* (C. Tschudin and J. Vitek, eds.), 1996.

[17] R. S. Gray, “D’Agents: Security in a multiple language, mobile-agent system,” in *Mobile Agents and Security* (G. Vigna, ed.), vol. 1419 of *Lecture Notes in Computer Science*, pp. 154–187, Berlin Heidelberg: Springer Verlag, 1998.

[18] N. M. Karnik and A. R. Tripathi, “Agent server architecture for the Ajanta mobile-agent system,” in *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA ’98)*, (Las Vegas), July 1998.

[19] C. Bryce and J. Vitek, “The JavaSeal Mobile Agent Kernel,” in *Proc. First International Symposium on Agent Systems and Applications, and Third International Symposium on Mobile Agents (ASA/MA ’99)*, 1999.

[20] G. Gzajkowski and T. von Eicken, “JRes: A resource accounting interface for Java,” in *Proc. ACM OOPSLA Conference*, (Vancouver, BC), October 1998.

[21] G. Karjoth, N. Asokan, and C. Gülcü, “Protecting the computation results of free–roaming agents,” in Rothermel and Hohl [24], pp. 195–207.

[22] V. Roth, “Secure recording of itineraries through cooperating agents,” in *Proc. 4th ECOOP Workshop on Mobile Object Systems: Secure Internet Mobile Computations*, (Brussels, Belgium), pp. 147–154, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex (France), Dépot légal 010598/150, July 1998.

[23] Sun Microsystems, Inc., *JDK 1.2 Documentation*, 1998. Available at URL: `http://java.sun.com`.

[24] K. Rothermel and F. Hohl, eds., *Proceedings of the Second International Workshop on Mobile Agents (MA ’98)*, vol. 1477 of *Lecture Notes in Computer Science*. Berlin Heidelberg: Springer Verlag, September 1998.

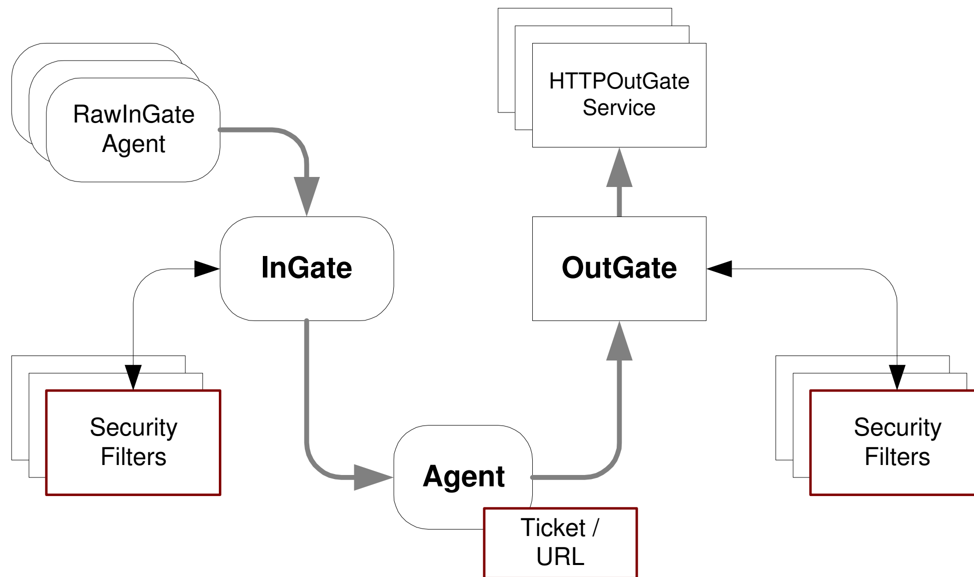

Figure 1. SeMoA supports multiple gateways for agent transport. Inbound and outbound gateway services are managed by the *ingate* and *outgate*; two central managements services which also pipe agents through any number of security filters before transport.

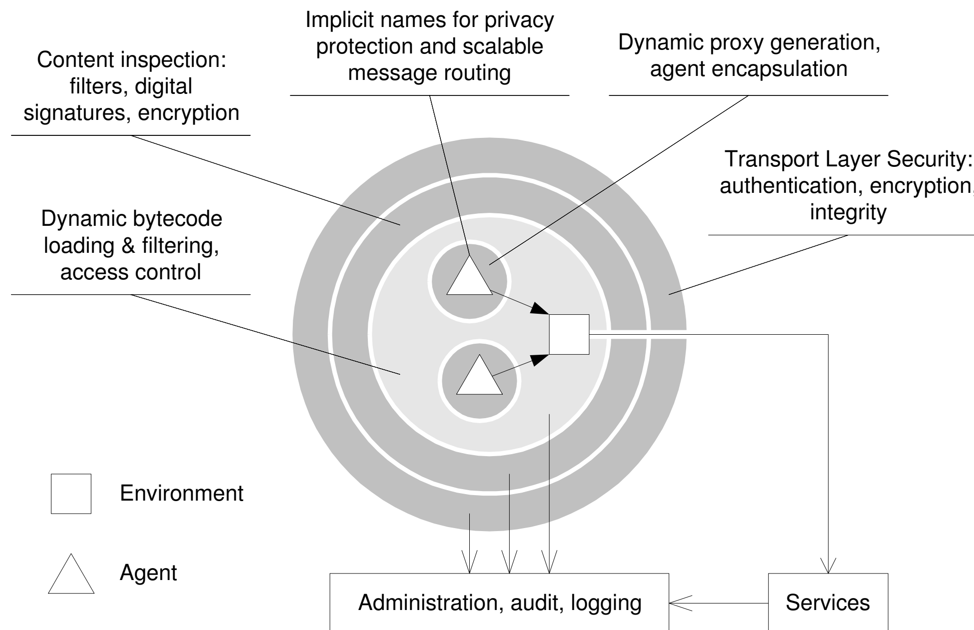

Figure 2. The general security architecture of SeMoA resembles that of an onion; agents have to pass all layers before being installed and launched in the server's runtime system.

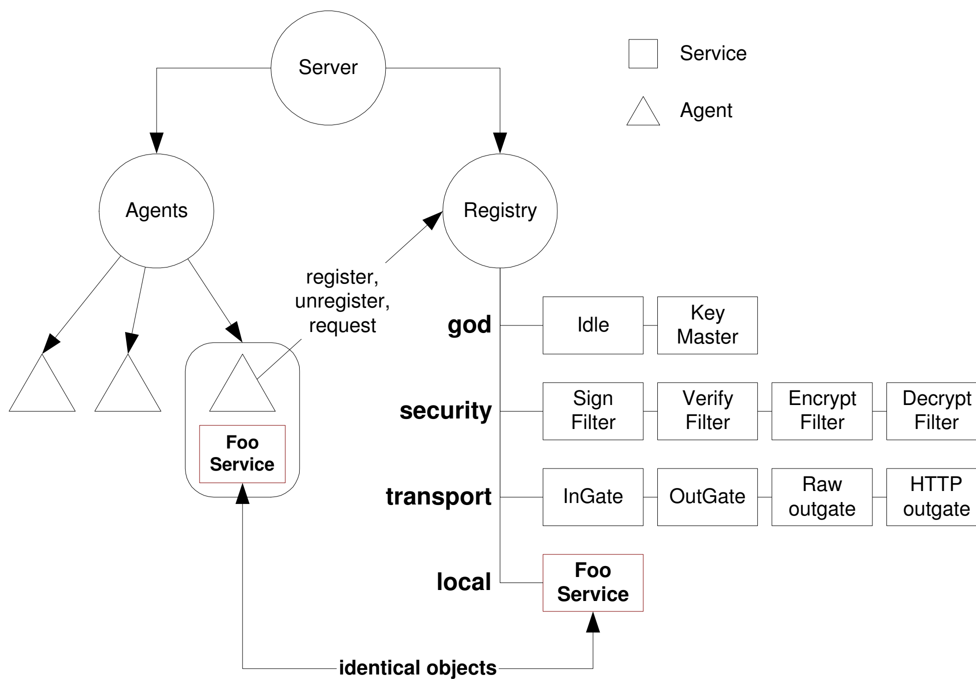

Figure 3. The general architecture of SeMoA distinguishes between agents and services. Agents can register, unregister, and request services. On the other hand, services can refer to and borrow resources from agents.