Proceedings of the International Multiconference on
Computer Science and Information Technology pp. 13–25

ISSN 1896-7094
© 2007 PIPS

A Common Base for Building Secure Mobile Agent Middleware Systems

Guido J. van ’t Noordende*, Benno J. Overeinder**, Reinier J. Timmer,
Frances M. T. Brazier, and Andrew S. Tanenbaum

Department of Computer Science, Vrije Universiteit Amsterdam,
Amsterdam, The Netherlands
{guido,bjo,rjtimmer,frances,ast}@cs.vu.nl

Abstract. The Agent Operating System (AOS) provides the basic functionality needed for secure and reliable mobile agent platforms: support for secure communication, secure agent storage and migration, and minimal primitives for agent life-cycle management. Designed as a layer between local operating systems and higher level agent platform middleware, it supports interoperability between agent platforms and between different implementations of AOS itself. AOS has been tested on interoperability, both with regard to different higher-layer middleware platforms and interoperability between two implementations of AOS in C++ and Java.

1 Introduction

Multi-agent system applications often rely on agent platforms (agent middleware systems) for agent life-cycle management, communication, possibly migration, and security [7, 1, 8]. Most (mobile) multi-agent systems to date are monolithic systems, where all functionality is integrated in a single code-base, often implemented in Java. Even if systems have a more or less modular design (e.g., JADE [1]), they are generally not designed for interoperability with other systems, or easily allow for integration of components written in different languages. Some solutions and specifications for interoperability exist (e.g., FIPA), however these specifications often define only higher-level functionality such as interagent communication protocols, and do not define low-level protocols for agent migration or setting up (secure) connections between middleware systems.

The system described in this paper, called Agent Operating System (AOS), is intended to facilitate designing mobile agent middleware systems in a more modular way, by providing a common, language-neutral base for layering multi-agent middleware systems upon. AOS offers a well-defined interface that provides primitives for secure packaging and migration of agents written in various languages, and for the establishment of secure channels between different mobile agent middleware components.

This paper proposes a multilevel architecture for agent middleware: a common minimal AOS layer, and higher level middleware layers for platform specific functionality.

* Current address: Informatics Institute, University of Amsterdam, The Netherlands
** Current address: NLnet Labs, Amsterdam, The Netherlands

The common minimal base, the “kernel” to higher level middleware systems, is the main focus of this paper. The requirements and design considerations for the AOS kernel are identified in Section 2. Section 3 presents the architectural design of AOS and evaluates the design with respect to the requirements. Section 4 evaluates two implementations of the AOS kernel: one in C and one in Java. Related work is discussed in Section 5, and the paper concludes with a summary in Section 6.

2 Design Requirements and Motivation

Most mobile agent middleware systems are designed to support specific agent models and programming environments. These systems share functionality. AOS has been designed to provide a minimal common base which provides this shared functionality to mobile agent middleware systems. By design, AOS supports interoperability, facilitating open and extensible design of agent middleware, and enabling interaction and/or integration with (existing) middleware services.

The commonalities found between agent middleware systems can be broadly classified as: (i) mobile agent (code and data) storage and transport, (ii) primitives for agent life-cycle management, and (iii) secure communication between middleware processes (irrespective of what is actually stored in an agent or being communicated). In addition, all current multi-agent systems require security mechanisms that allow for, e.g., authentication and authorization of remote processes, and for integrity verification of migrated agents and content.

2.1 Requirements

To realize a common base for implementing secure and modular mobile agent middleware, the following requirements for the AOS kernel are defined:

- AOS should be a common layer for a broad range of agent middleware systems.
- The AOS design and specification should be language and operating system neutral, so that it can be implemented in any programming language and ported to any operating system. All these implementations should be able to interoperate.
- AOS should be minimal, in that it should only provides the minimal set of primitives required for building (secure) mobile agent middleware. Minimality ensures that the AOS code base becomes manageable and can be implemented in a robust and secure way. This implies that some mechanisms have to be implemented by the agent middleware itself, which is inherent to the idea that such mechanisms are middleware specific. In short, AOS should be “lean and mean”, and provide only the basics needed for implementing (secure) mobile agent middleware.
- AOS should be reasonably efficient, within the expected performance boundaries of (secure) agent middleware. In particular, it should not add significant overhead compared to a distributed mobile agent system written from scratch, assuming that such a system is built with comparable (security) requirements in mind.
- AOS should not impose design limitations or a specific model on the mobile agent middleware designer. For example, it should not require the designer to adopt a specific deployment or security model (e.g., using a specific public key infrastructure).

- The AOS should be a stand-alone component which can be compiled and used without administrative privileges on a hosting machine.
- AOS should be usable by different mobile agent middleware systems concurrently, although stand-alone (non-shared) usage should also be possible.

2.2 Motivation of Design Requirements

The requirements outlined above have several important implications for the design and implementation of a minimal kernel. This section describes the rationale for the requirements and discusses some of the consequences.

Making AOS language-neutral is an important requirement to provide a support layer for agent middleware. Most current agent middleware systems have been implemented in Java, and consequently support Java as a programming language for agent development. However, this may not always be the most obvious programming language for agent development; C++ or Python may be preferable for the task at hand, maybe for code reusability, for interoperability, or for performance reasons.

The minimality and reasonable efficiency constraints have implications for interactions with the underlying system and management of resources. AOS only support local interactions, i.e., local processing accessing local resources, which should not block or interfere with other (local) operations. The only non-local interaction is communication. Restricting AOS to managing local resources only, avoids having the kernel wait for remote services to answer before it can complete a task. Management tasks spanning more than one machine are the responsibility of the higher-level middleware systems.

Security is very important in mobile agent systems, both from the perspective of the agent as well as of the host. As mobile agents move to foreign hosts (which may not always be trusted or trustworthy), their data and code should be protected from tampering. Mechanisms must be provided for timely agent tampering detection, and for authentication of remote agent middleware (see Section 3.2).

From a host's perspective, mechanisms are needed to protect hosts from malicious or erroneously programmed agents. Sandboxing (for interpreted executables) or jailing (for binary executables) [9], two examples of mechanisms that allow for protection of a host from malicious agents, require interaction with higher-level middleware systems in which life-cycle management is regulated. AOS provides mechanisms for agent code and data management, which provide the basis for implementing agent life-cycle management in the middleware layer.

The requirement that AOS should be usable by multiple mobile agent systems at the same time allows for AOS to provide an access point to multiple mobile agent middleware systems behind a firewall, among other things. The number of TCP ports for incoming connections should therefore be minimal. Sharing one AOS instance on a host requires an authentication mechanism for agent middleware to allow separation of operation. This mechanism is discussed in Section 3.

3 Architecture of the AOS Kernel

This section discusses the architecture and design of the AOS kernel.

3.1 Architectural Model of AOS

The intended use of AOS is to provide a common base to a range of specific *Mobile Agent Middleware (MAM)* systems. This common base can be seen as a kernel component in a layered middleware system design. Higher-level agent middleware systems use AOS for agent code and state management, agent migration, and communication. Higher-level MAMs extend the basic common AOS layer with agent middleware specific components and possibly one or more services, e.g., for agent life-cycle management, middleware management, and agent naming and location services. The general architectural model is shown in Fig. 1.

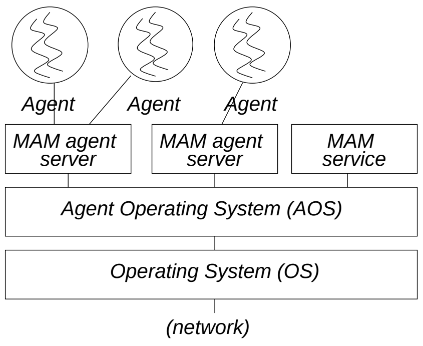


Fig. 1. Example of a layered agent middleware architecture using AOS. This example system consists of two agent server processes and one service (e.g., a naming service) running on the same machine. MAM processes communicate with other local or remote MAM components using AOS. Agents communicate with their runtime environment (e.g., agent server), and do not normally access AOS directly.

AOS provides a means for middleware processes to securely authenticate services and other MAM components in a system, to communicate with these components and services, and to migrate agents to other locations in a secure way. AOS has two external entries for receiving agents and for incoming connections to middleware processes that use AOS. How authentication, communication, and agent transport are implemented internally in AOS is described later in this section.

MAM components (distinct processes from an architectural point of view, see Fig. 1), are responsible for the agent runtime environment. Agents are executed by a MAM component, which provides a runtime environment (API) to them with MAM specific functionality. The MAM components use AOS internally when they communicate with each other, when an agent's state needs to be altered, or when it needs to be migrated to another AOS kernel. Note that services do not host agents (see Section 3.2).

AOS has a specification that clearly describes the methods (API) provided by AOS to higher-level middleware systems, including arguments and semantics. The AOS kernel effectively hides differences in the underlying operating system with regard to communication interfaces and file system access from the processes that use AOS. MAM components invoke AOS API methods using RPC calls. Different RPC interfaces have been implemented, for Sun RPC, Java-RMI and XML/RPC. This allows the MAM processes to be implemented in a different language than AOS, even within the same MAM. AOS allows for multiple so-called dispatchers that implement different RPC implementations, to be used simultaneously such that any component can use an RPC interface of its choice.[1]

3.2 AOS Concepts and Primitives

The AOS API provides primitives for agent transport (shipping/receiving ACs), and communication (creating endpoints and connecting/accepting connections). In addition, the API contains calls allowing for safe sharing of AOS between different MAM components. The agent transport mechanism provides integrity protection of agent code and (meta-) data, and the communication methods include a simple but highly effective authentication model (see Section 3.3). Note that the API contains no primitives for process management. This is because different mobile agent middleware systems have very different methods for managing agent processes. For example, some agent systems use a thread-based model, where agents run as a thread in an agent server process, whereas in other systems each agent runs as an independent process. As a result, it is hard to attain a single, simple model for agent process management, and process management is therefore left to the mobile agent middleware.

Agent Containers Agent code, data, and meta-data (e.g., owner information, time of creation, permissions, etc.) are stored in AOS in a data structure called the *Agent Container (AC)*. The AC is, in fact, an archive with a table of contents, immutable segments (e.g., code), and mutable segments (e.g., state and auxiliary data). Primitives for creation of AC and segments, and reading/writing segments are part of the API.

In addition, a *finalize* call is used to synchronize any new content of the AC to disk (to allow for recovery if AOS crashes and is restarted[2]), and to generate checksums in a secure *Table of Contents (ToC)* of the AC's segments. The secure ToC is used for integrity verification of an AC when it is received by another AOS kernel. Finalize must be called before an agent can be migrated to another AOS.

Communication Endpoints Communication-related calls include creation and deletion of communication endpoints (similar to Unix sockets). Connect, accept, send, receive, and select calls exist which allow for setting up and using secure, reliable, ordered communication channels to these endpoints. Connections between the same pair of AOS kernels, with the same security properties (i.e., cryptographic *cipher suites*), are multiplexed over a single AOS "base channel" internally to AOS. This allows for

[1] In particular, it is straightforward for a Java-based middleware to use Java-RMI, whereas a C-based middleware implementation can more straightforwardly use Sun RPC.

[2] The API defines error codes that allow for detection of an AOS restart, and contains a call for re-initializing AOS resources after such an event was detected by the middleware.

a reduction of connection setup times by amortizing expensive initial secure connection setup times over multiple connections, compared to setting up new secure connections for each connection. The method for setting up secure connections is discussed in Section 3.3. Agent transport makes use of the same internal AOS connection implementation, allowing for safe (integrity and confidentiality protected) transport of agent containers.

Secure Sharing of AOS Secure sharing of a single AOS instance and its resources by different unrelated agent middleware on the same host is enabled by the concept of a *role*. A role is a set of resources associated with a cryptographically protected authentication token (called a *cookie*), which is used by an agent middleware to invoke methods on AOS. AOS creates this cookie securely, and also creates an internal data structure that describes the resources associated with this cookie. We refer to this cookie and its associated resources as a role.

During start-up of AOS, an *init role* is generated, that is allowed to invoke any method on the AOS API. The init role is used by an *init process* that controls usage of AOS (think of *init* process in Unix). Given the init role, the init process can generate other roles, which are called *child roles*. Generally, these roles allow users to create additional subroles for components within the same agent middleware.[3] For example, a service may only be allowed to use communication related calls, and agent servers may only be allowed to access the ACs of the agents they manage. Assigning a new role to each MAM component allows for compartmentalization of an agent middleware system. This avoids that a single compromised MAM component can compromise the state of another MAM component that uses the same AOS instance, e.g., by destroying AOS kernel objects such as ACs.

Roles determine ownership of resources in AOS. Resources include agent containers, communication channels, and child roles. When a role is deleted (using an AOS call), all resources associated with this role are deleted, including subroles and their resources. In principle, roles are persistent such that role information and resources (in particular, ACs) can be recovered after an AOS crash or system reboot. AOS implements resource protection by imposing limits on the number of resources (and subroles) owned by a role.

3.3 Authentication Model

AOS comes with a simple but highly effective authentication model based on public key cryptography, that is used when connections are set up and agents are shipped to other AOS kernels. The authentication model is based on the concept of *Self-certifying Identifiers (ScIDs)* [5]. A ScID is a SHA-1 hash of the public key of an AOS kernel, where the AOS kernel has access to the associated private key.

Each endpoint created by an AOS kernel, both for AC transport and for communication, is described by a data structure that contains a ScID in addition to the AOS kernel's endpoint information (i.e., IPv4/v6 address and port). This data structure is used by a

[3] Each user may obtain its own initial role by requesting it from the init process, or through some other way; the way in which an init process functions precisely is not currently specified.

MAM component to set up a connection or to ship an AC to another MAM component using AOS, where AOS internally verifies that its peer AOS kernel has the private key corresponding to the ScID in the AOS contact record. A standard authentication protocol (i.e., TLS/SSL) is used for authentication and key-exchange, as part of setting up an efficient, secure, encrypted channel to the peer AOS. The MAM component can specify the cryptographic cipher suite for the channel at connection setup time. The advantage of ScIDs over, e.g., X.509 based authentication models, is that no PKI infrastructures are required to bind keys to identities, as ScIDs are used as the name of the entity (AOS kernel), and are coupled directly to its key as described above.

3.4 End-to-End Authentication and Secure Communication

When AOS is shared between multiple MAM systems, each MAM should be able to authenticate its peer MAM end-to-end, to ensure it is not connected to another MAM that uses AOS at the same time. To this purpose, the AOS endpoint data structure explained above can be included in a *Middleware Contact Record (MCR)*, in addition to information that can be used to securely identify (authenticate) the middleware running on top of AOS. This MCR can be used by the MAM to set up a secure, end-to-end authenticated connection to another MAM, using an encrypted transport set up using the AOS contact information in the MCR. An MCR can contain a Self-certifying ID (ScID) of the peer middleware process, so that it can be authenticated using the mechanism outlined in Section 3.3. Another approach is to use a simple name in the MCR which can be combined with a PKI to securely authenticate a remote middleware process. AOS does not force any particular authentication model upon the middleware.

To securely use AOS channels as the basis for an end-to-end authenticated channel, both middleware processes must, after authenticating each other, exchange authenticated messages to each other that contain the AOS endpoint information of their own (trusted) AOS kernel. This is required, because if this check does not take place, an impostor AOS kernel may sit between the endpoint AOS kernels as a man-in-the-middle, which could decrypt and read all information passed over the channel—as generally only MAM authentication information is known at the time when a connection is made. After such information is exchanged, both parties are certain that they communicate through the remote AOS kernel that is actually used by their peer. Only in this case, can the confidentiality of the underlying AOS channel be trusted, allowing the middleware processes to let AOS take care of encryption of the connection itself without the middleware processes having to encrypt the connection end-to-end.

3.5 Secure Agent Migration

AOS provides mechanisms for shipping an AC to another AOS kernel. AOS signs a secure *Table of Contents (ToC)* of the AC, which is used by the receiving AOS kernel for integrity protection. The ToC data structure is available to the MAM layer. The MAM layer has to implement a secure agent transport protocol (ATP) on top of the AOS mechanism for shipping an AC to allow for additional, middleware specific authentication and control over agent transport. For example, specific code segments may have to be present in an AC as a prerequisite for starting it up in a specific MAM process. Other extensions to the basic ATP provided by AOS are the construction of secure audit trails, describing all changes made to the AC during the agent’s itinerary [8, 10]. Different security mechanisms at the MAM layer can be conceived (see, e.g., [10]); AOS provides all the necessary hooks to construct such mechanisms effectively.

4 Performance

For a central component such as AOS, which is intended to be used for all interprocess communication, mobile agent code/data management and migration operations, performance is highly important. This section presents the approach in which two AOS kernels were implemented and used in our department, and discusses performance of these AOS kernels. Measurements of performance of agent middleware that runs on top of AOS is not provided in this section; since these aspects are middleware implementation specific, it is chosen to focus on the overhead of functionality of AOS itself, which is most likely to impact the performance and scalability of the agent middleware system that uses AOS. In particular, communication throughput and scalability have been tested, and AC shipment related overhead and scalability in terms of concurrently shipped ACs.

Independently, two versions of AOS have been implemented, one in Java and one in C++, based on a precise specification of the AOS interface and the internal protocols used by AOS.[4] These AOS kernels have been thoroughly tested on interoperability. Both the Java and the C++ kernel are used to construct two different agent middleware systems in our group, Mansion [10] (written in C) and AgentScape [6] (written in Java). These two mobile agent systems are quite different in their design and implementation decisions; even so, AOS has shown to be a solid basis for their construction.

This section evaluates the performance of the Java and C++ AOS kernels. Although the tests are limited to AOS operations, the aspects measured are expected to most influence the performance of a mobile agent middleware system built using AOS.

All tests were run on a dedicated 1 GHz dual Pentium-III machine with 1GB memory, running Linux on an ext3 filesystem and using a Fast Ethernet (100 Mbit/s) local area network. Tests with the Java kernel used the Sun Java 1.5 standard compiler and Java HotSpot server virtual machine version 1.5. The cryptographic libraries used in the Java AOS kernel are from Bouncy Castle.[5]

The tests were run with modified AOS kernels that included microsecond timers, and were executed 5 to 10 times in a row, with averages shown in this section. For all tests that use AOS-to-AOS communication, the connection was configured to use 128 bits AES encryption with SHA-1 message authentication.[6]

4.1 AOS-to-AOS Communication Cost

AOS uses an internal protocol to multiplex communication channels over a single internal encrypted “base channel.” Figure 2 shows the performance and scalability of AOS

[4] The AOS specification can be requested from the authors.
[5] `http://www.bouncycastle.org`
[6] AES provides a reasonable trade-off between security and efficiency, compared to e.g., 3DES.

for communication, for 1 to 16 threads communicating concurrently over AOS. In this experiment, each thread sends 25 MB over an AES encrypted base channel to a server process running on a different AOS kernel.

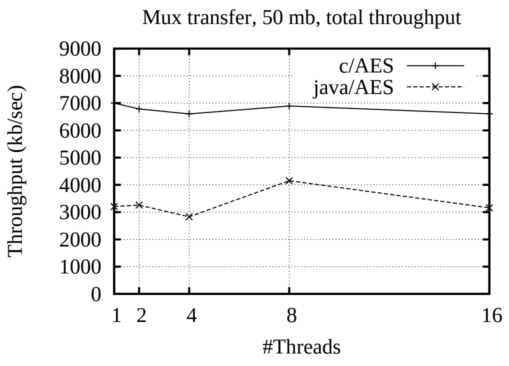


Fig. 2. Total throughput for multiplexed communication over a shared AOS-to-AOS connection.

As shown in Fig. 2, the C++ kernel has a substantially higher throughput than the Java kernel, which is due to the fact that the OpenSSL library implemented in C is faster than the pure Java Bouncy Castle SSL implementation used in the Java kernel.[7] Both kernels apply locking strategies to make sure that only a single thread can write payload on the base channel. The figure shows that the total throughput stays roughly the same for both kernels, irrespective of the number of threads that simultaneously send payload over the wire, although some variation exists which remains unexplained. Although the per-thread throughput decreases linearly with the number of threads for obvious reasons (i.e., sharing and overall saturation of the underlying connection), the AOS kernels and the internal protocol used for multiplexing do not adversely influence scalability.

4.2 Finalize Costs

Prior to shipping an AC, an AC must be finalized to ensure that the AC's table of content is generated, and that all segments are stored safely in a zip file synchronized to disk. Finalize is a call that constructs a secure Table of Content of the AC and signs it, prior to shipping it to another AOS kernel. In addition, finalize syncs the AC to disk for crash recovery reasons.

Table 1 shows a microbenchmark of the finalize costs of agent containers of 500 KB, 1 MB and 5 MB containing random data. These sizes are typical for many agents used in our own agent middleware system. ToC checksumming and signing cause little overhead, even for large ACs, and this increases linearly with the size of the AC. This is because the checksum (SHA-1 hash) generation has to take place over every byte of all

[7] Performance measures with an unencrypted (NULL) SSL channel show that Java performance in the unencrypted case comes close to the performance of the C++ kernel in the same scenario.

Table 1. Finalize micro benchmarks (in milliseconds) for resp. the C++ kernel and the Java kernel.

	C++			Java		
	500kb	1mb	5mb	500kb	1mb	5mb
checksum	9	19	98	36	74	70
sign	51	52	70	5	16	51
zip	133	248	1356	145	303	1449
sync	166	238	922	179	401	1623
total	359	558	2446	442	878	3854

segments. Creating a zip file and synchronizing it to disk cause substantial overhead, because zipping requires that each segment is copied into the zip file, possibly after compression. Synchronizing the resulting zip file to disk is also rather expensive. Finalize times scale roughly linearly with the AC sizes for both the Java and the C kernel, although finalize takes substantially longer on the Java kernel than on the C kernel.

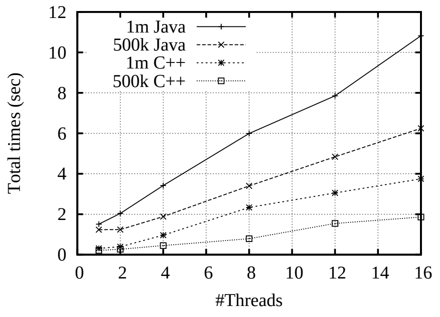
As mobile agents may migrate often during their lifetime, AC finalize and transfer cost can increase the time for an agent to achieve its task considerably, and influences scalability of the mobile agent middleware as a whole, which is confirmed by experiences with our own agent middleware systems. A straightforward optimization for performance, is to have AOS ship segment files to another AOS kernel directly, without zipping the files first, in an FTP-like manner. Another possible optimization is to let go of the crash recovery assurance by means of the fsync system call.

4.3 AC Shipment Cost

AC shipment is composed of a `ship_ac` primitive combined with a `wait_ac` primitive at the receiving end, which returns after shipment is completed. Ship_ac takes a finalized agent container, and ships it over an SSL connection as described above. After receipt of the AC, the receiving AOS kernel extracts the agent’s zip file containing the agent’s segments, and verifies the checksums in and the signature over its ToC. Only after this verification, wait_ac returns. After an acknowledgement is received for all shipped ACs, the timer is stopped. The ship_ac cost measured thus includes both the on-the-wire time and the extract/verify cost at the receiving end.

The total ship_ac costs including AC extraction and verification were measured for AC sizes of 500 KB, 1 MB, and 5 MB containing identical segments of 5 KB random binary data. The cost of extracting and verifying an AC after it is received depends primarily on the size of the AC. The times for ZIP extraction (default compression ratio) and signature and checksum verification in the C++ kernel are 0.064, 0.127, and 0.734 sec. for 500 KB, 1 MB, and 5 MB, respectively. Extraction and verification in the Java kernel takes substantially longer, namely 0.597, 1.033, and 3.472 sec., respectively. Of these times, about 80–90% is spent on unzipping the AC.

Figure 3 shows the results for both the C++ kernel and the Java kernel for 1, 2, 4, 8 and 16 ship_ac calls at the same time. The figures show that AOS ship_ac calls scale roughly linearly with concurrent use. The figures also show that the time needed to ship an AC is more for the Java kernel than for the C++ kernel. This can be attributed in



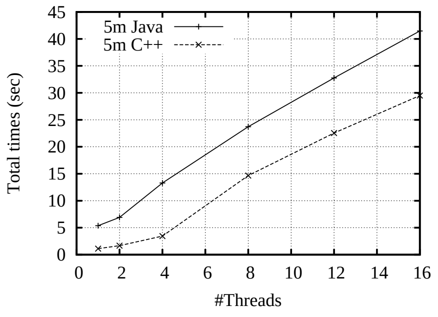


Fig. 3. Elapsed time to ship 1–16 Agent Containers of 500 KB resp. 1 MB (left), and 5 MB for both the Java and the C++ kernel.

part to the fact that cryptography (for encrypting the connection) and AC extraction and verification take longer in Java than in C++.

5 Related Work

Comparison of AOS with other related work should be done with the design requirements of AOS in mind. As AOS is not an agent middleware itself, but rather a middleware building block, comparing AOS with full functional middleware can only be done partially by considering the leading design requirements.

The FIPA standard specification includes a series of documents describing the functionality and operation of agent middleware. FIPA compliant agent middleware can interoperate which each other, e.g., agents can exchange messages, interact with, and reason about agents on other middleware. One of the most widely used FIPA compliant agent middleware is JADE [1]. The latest middleware design (version 3.5 as of today) is modular in design and many parties (universities and companies) have contributed to JADE. The middleware is implemented in Java and supports a Java API for agent development. It is a complete self-relying system, with integrated location and yellow pages services. This is different from the AOS perspective to agent middleware, where services can be arbitrary location or yellow pages services such as DNS or LDAP servers.

Ajanta [4] is designed to include a number of security primitives and architecture features to protect both the host and the agent from malicious actions. It includes amongst others a similar concept as the agent container in AOS, allowing for an audit trail mechanism resembling the one outlined in this paper and in [10]. However, Ajanta is completely Java-based and is not designed to incorporate or interact with other software components or services.

The Tacoma [3] project focuses on operating system support for mobile agents. In that respect, it has many similar design goals as AOS by providing abstractions for, in particular, data storage and agent mobility. Although it also provides a simple container abstraction, called a “briefcase”, only very simple protection mechanisms were implemented. Tacoma supports multiple programming languages for agents, in particular C and Tcl/Tk.

The MadKit agent platform architecture [2] aims to provide a generic multi-agent platform. The architecture is based on a minimalist agent kernel decoupled from specific agency models. Although there are similarities with the design goals of the architecture model with AOS, the design and implementation is quite different. The aim of MadKit is to allow a developer to implement its own agent architectures. Basic services like message passing, migration, monitoring, or management, are provided by platform agents. MadKit comes with a set of "containers", realizing different execution environments for running an application. Alternatively, AOS aims to provide a layer for constructing different agent middleware, and is not directly used by agents.

6 Summary

This paper discusses the design requirements, implementation, and performance of the AOS kernel. AOS is a portable middleware building block specifically aimed at constructing mobile agent middleware systems. It can be used by different MAM processes, possibly of different users, independently, where each such process may be implemented in a different language. Programming language flexibility is facilitated by the use of different RPC dispatchers, each providing a method invocation interface suitable for a specific language. The AOS design allows for secure sharing of a single AOS kernel between different middleware processes: it provides effective software fault isolation and safety by separating resources created by different middleware processes.

AOS provides a minimal set of primitives that are general to mobile agent systems, in particular for agent code and data storage, agent transport, and for communication between mobile agent middleware components. AOS provides basic security services which can be used by higher-level middleware layers to construct more elaborate security, such as authentication mechanisms and secure agent transport and auditing of mobile agents. AOS does not impose a specific model on the agent middleware.

Two implementations of AOS have been built and tested for interoperability. Performance measurements of the AOS kernel were shown in this paper, which show that AOS performs reasonably well, with noticeable differences between the Java and the C++ kernel. The C++ kernel outperforms the Java kernel for most tests, primarily due to the fact that C is more efficient than Java for tasks such as cryptography, which is used throughout the AOS kernel. On the other hand, Java provides better portability, and the Java kernel has been used to run the AgentScape mobile agent platform on Linux, Solaris, Mac OS X, and Windows systems. The C++ kernel is currently only available for Linux and Solaris platforms. Both implementations of AOS were shown to scale well with respect to concurrent usage by middleware systems for communication and transport of Agent Containers, which is important when using AOS to construct large-scale distributed mobile agent systems.

Acknowledgements

A number of colleagues have contributed ideas and helped to code parts of the AOS kernel. The authors would like to acknowledge Etienne Posthumus, Patrick Verkaik, Arno Bakker, David Mobach, and Michel Oey. Maarten van Steen and Niek Wijngaards are acknowledged for early contributions to this work. This research is supported by the NLnet Foundation, `http://www.nlnet.nl`

References

1. F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Software – Practice and Experience*, 31(2):103–128, 2001.
2. Olivier Gutknecht and Jacques Ferber. The MADKIT agent platform architecture. In *Proceedings of the International Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55, Montreal, Canada, June 2000.
3. D. Johansen, R. van Renesse, and F.B. Schneider. Operating systems support for mobile agents. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 42–45, Orcas Island, WA, May 1995.
4. N. Karnik and A. Tripathi. Security in the Ajanta mobile agent system. *Software – Practice and Experience*, 31(4):301–329, April 2001.
5. D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, 1999.
6. N. J. E. Wijngaards; B. J. Overeinder; M. van Steen; F.M.T. Brazier. Supporting Internet-Scale Multi-Agent Systems. *Data and Knowledge Engineering 41(2-3)*, 2002. pp. 229-245.
7. N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith. NOMADS: Toward a strong and safe mobile agent system. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 163–164, Barcelona, Spain, June 2000.
8. Anand R. Tripathi, Tanvir Ahmed, and Neeran M. Karnik. Experiences and future challenges in mobile agent programming. *Microprocessor and Microsystems*, 25(2):121–129, April 2001.
9. G. J. van 't Noordende, A. Balogh, R. Hofman, F. M. T. Brazier, and A. S. Tanenbaum. A secure jailing system for confining untrusted applications. *International Conference on Security and Cryptography (SECRYPT), Barcelona, Spain*, July 28-31 2007.
10. Guido J. van 't Noordende, Frances M. T. Brazier, and Andrew S. Tanenbaum. Security in a mobile agent system. In *Proceedings of the First IEEE Symposium on Multi-Agent Security and Survivability*, pages 35–45, Philadelphia, PA, August 2004.