Constructing secure mobile agent systems using the agent operating system

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

Department of Computer Sciences, Vrije Universiteit Amsterdam,
de Boelelaan 1081, 1081HV, Amsterdam, The Netherlands
E-mail: guido@cs.vu.nl
E-mail: bjo@cs.vu.nl
E-mail: rjtimmer@cs.vu.nl
E-mail: frances@cs.vu.nl
E-mail: ast@cs.vu.nl
*Corresponding author

Abstract: Designing a secure and reliable mobile agent system is a difficult task. The agent operating system (AOS) is a building block that simplifies this task. AOS provides common primitives required by most mobile agent middleware systems, such as primitives for secure communication, secure and tamper-evident agent packaging and agent migration. Different middleware processes can use AOS at the same time; effective security mechanisms protect AOS resources owned by different middleware processes. Designed as a portable and language-neutral middleware layer residing between the mobile agent system and the operating system, AOS facilitates interoperability between agent platforms and between different implementations of AOS itself. AOS has been implemented in both C++ and Java. This paper motivates the design of AOS, describes how AOS is used in a mobile agent system, and presents performance measures for an agent transfer protocol layered upon AOS.

Keywords: secure agent middleware design; mobile agent systems; security; agent transfer protocol; ATP; audit trails.

Reference to this paper should be made as follows: van 't Noordende, G.J., Overeinder, B.J., Timmer, R.J., Brazier, F.M.T. and Tanenbaum, A.S. (2009) 'Constructing secure mobile agent systems using the agent operating system', *Int. J. Intelligent Information and Database Systems*, Vol. 3, No. 4, pp.363–381.

Biographical notes: Guido van 't Noordende is currently finishing his PhD work on the Mansion mobile agent system, which he designed at the Vrije Universiteit Amsterdam. He is currently affiliated with the University of Amsterdam where he does research on security of distributed systems, in particular, on security and privacy in grid systems.

Benno Overeinder received his PhD at the University of Amsterdam and is currently a Senior Researcher at NLnet Labs, a non-profit R&D organisation that develops open standards and open source software for internet protocols. His former affiliation was Assistant Professor in the IIDS group at the Vrije Universiteit Amsterdam.

Reinier Timmer received his MSc from the Vrije Universiteit Amsterdam. He was a Scientific Programmer in the IIDS Group, and is currently working for Thales Netherlands still working on and with agent middleware systems.

Frances Brazier was a Full Professor at the Vrije Universiteit Amsterdam before she and the IIDS Group moved to the TU Delft, where she is a Full Professor now. Her group's research focuses on (self-) management of large scale distributed autonomous systems. AgentScape is a mobile agent middleware designed to support large scale distributed autonomous systems.

Andrew Tanenbaum is a Full Professor at the Vrije Universiteit Amsterdam and heads the Computer Systems group, which does research on distributed systems, operating systems and security. He is the author of a number of widely used books on operating systems, networking and distributed systems.

1 Introduction

Over the last decade, various mobile agent middleware systems – also called mobile agent systems or mobile agent platforms – have been developed to support (mobile) multi-agent applications (White, 1996; Baumann et al., 1997; Johansen et al., 1995; Suri et al., 2000; Bellifemine et al., 2001; Tripathi et al., 2001). Mobile agent applications depend on middleware mechanisms for agent life-cycle management, communication, migration and security.

Existing mobile agent systems were designed with different goals and foci, e.g., on communication, mobility, security, agent model support, management, etc. (Milojicic et al., 1999). Most existing agent platforms were implemented as monolithic systems, where all functionality is integrated in a single code-base. Even if a system has a more or less modular design (e.g., JADE, Bellifemine et al., 2001), it is generally not designed for interoperability with other systems or to easily allow for integration of components written in different languages. For example, over time, various mechanisms for security were designed, such as audit trail based security (Tripathi et al., 2001), but few of these mechanisms were adopted by other mobile agent systems. Interoperability specifications like FIPA or MASIF define higher-level functionality such as inter-agent communication protocols. However, these specifications do not define how low-level protocols for, for example, agent migration or communication should be implemented.

Most agent systems have been implemented in Java, which provides portability and some security assurance. Very few systems support agents written in other programming languages than Java (Gray et al., 1998). However, for many tasks, support for agents in different languages is useful. For example, legacy programs may have to be re-used in mobile agents. Also, middleware implementations may benefit from using different programming languages internally. For example, some parts of a mobile agent system may be implemented in C for performance, while other parts may be implemented in Python for rapid prototyping.

Based on our own experience in constructing two different mobile agent systems (Wijngaards et al., 2002; van 't Noordende et al., 2004), we identified a minimal set of common primitives required by mobile agent middleware systems. Rather than focusing on solutions for a specific middleware system, we decided to construct a generic 'kernel',

called Agent Operating System (AOS), which provides the common primitives required for constructing mobile agent systems – including our own, but also those known from related work.

AOS is a common minimal base for constructing higher-level middleware systems. It has been implemented in C++ and Java using a single specification of the AOS API and the low-level protocols.[1] These two implementations were intensively tested for interoperability. A performance comparison between the Java and the C++ implementation was given in an earlier version of this paper (van 't Noordende et al., 2007b). In this paper, we concentrate on the C++ implementation for brevity.

AOS is specifically designed as a stand-alone component that can be shared between multiple different middleware components (processes) running on the same machine. AOS resides in a separate address space from other middleware processes. Having AOS run in a separate address space allows for effective protection against faulty or compromised middleware components. An access control mechanism guards access to data stored in AOS and mediates access to AOS primitives. AOS provides language-independent access to its methods and data structures using one or more RPC interfaces. This allows different middleware components written in different programming languages to access the methods and data structures provided by AOS.

Besides interoperability and language-independence, a leading requirement when designing AOS was security. AOS provides highly secure agent packaging and shipment primitives, as well as secure communication primitives that allow for establishing authenticated communication channels to remote middleware processes. AOS also provides an efficient mechanism for constructing audit trails (Karnik and Tripathi, 2001), that allows to detect tampering with an itinerant agent that has migrated over multiple machines.

The paper is organised as follows. The requirements and considerations that drove the design of AOS are described in Section 2. Section 3 presents the architectural design of AOS in detail. Section 4 describes the design of a secure mobile agent system which was built on top of AOS. Section 5 evaluates the performance impact of using the AOS kernel written in C++ for implementing an agent migration protocol. Related work is discussed in Section 6, and the paper concludes with a discussion in Section 7.

2 Design motivation

With the design and implementation of most agent middleware systems, a set of goals drive the development process. Most notable are the support for specific agent models, programming environments, mobility and security. Although the design goals and implementation decisions of mobile agent middleware systems differ, all systems have some basic functionality in common. AOS has been designed to provide this common functionality in a modular and secure way, to support a wide range of conceivably very different agent systems.

The commonalities found between agent middleware systems can be broadly classified as:

1 mobile agent (code and data) storage and transport

2 primitives for agent life-cycle management

3 mechanisms for (secure) communication.

In addition, all current multi-agent systems require security mechanisms that allow for authentication and authorisation of remote processes, and for integrity verification of migrated agents and content.

AOS should not impose design limitations or a specific model on the mobile agent middleware using it. For example, AOS should not require the middleware designer to adopt a specific programming language or security infrastructure. In short, AOS should be 'lean and mean' and provide only the basics needed for implementing (secure) mobile agent middleware, but nothing more. This implies that some mechanisms remain to be implemented by the agent middleware itself, which is inherent to the idea that many mechanisms are middleware specific. Minimality also ensures that the AOS code-base becomes manageable and can be implemented in a robust and secure way.

We decided that 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. AOS should provide language-independent access to its methods. The solution we chose is that AOS provides one or more RPC interfaces to expose its methods to the middleware, which are accessed by middleware processes using simple RPC stubs (Section. 3.1). RPC provides highly effective fault isolation: a breach in a client process cannot directly spread to the code of the AOS kernel. This is an important reason for designing AOS as a process that runs in a separate address space from other middleware processes. By providing several RPC interfaces at the same time, a mobile agent designer can choose to use different languages for implementing different middleware components, possibly even different components that run at the same time.

It is important that AOS itself does not rely on external services, such as location services. Such reliance could hurt reliability and performance, or possibly even hinder other middleware processes that use AOS, since interactions with a remote process can block or fail in several ways. Management tasks spanning more than one machine are the responsibility of the higher-level middleware system. AOS interacts with other AOS processes only if this is required by the agent middleware, e.g., for setting up a communication channel or when shipping an agent.

We consider it convenient that AOS is usable by different middleware systems, possibly owned by different users, at the same time. An example is where a single AOS kernel is started up at system boot time, to which different middleware processes on this system can connect. The advantage of sharing a single AOS kernel between agent middleware systems is that there is a need for opening only one or two TCP ports in the system's firewall. Using an AOS kernel that resides at these ports, different middleware processes can communicate and ship agents to other middleware processes, without requiring separate ports to be opened for each middleware system or application. Because AOS may be shared between different middleware processes, it is particularly evident that AOS must isolate the resources and data of different middleware processes. An efficient and flexible access control mechanism is devised to separate AOS resources owned by different middleware processes. The same mechanism is used to implement secure *internal* compartmentalisation of middleware systems (Section 3.2.3).

Security is very important in mobile agent middleware, 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. AOS comes with an efficient agent data and code integrity verification mechanism. On top of this mechanism, agent middleware can implement an efficient

audit trail verification mechanism, which helps protect the integrity of agents that migrate over multiple hops (Section 4).

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) (van 't Noordende et al., 2007a), are two examples of mechanisms that allow for protection of a host from malicious agents. However, the way in which mobile agent middleware systems handle host protection and agent lifecycle management differs widely. For example, some agent systems use an agent server, in which agents are started up as threads, while other agent systems start up each agent as a separate process. As a result, it is hard to attain a single, simple model for secure agent execution and lifecycle management. For this reason, we decided to leave agent lifecycle management mechanisms to the agent middleware to implement, and not to provide agent lifecycle management mechanisms in AOS.

AOS provides a mechanism for authenticating a remote AOS kernel as part of setting up a secure, reliable, ordered communication channels. We chose for a channel abstraction because, when secrecy is required, cryptographically protected communication channels can be implemented much more efficiently than when a message oriented approach is used. If an agent middleware requires this, (reliable, ordered) messaging primitives can be layered straightforwardly upon the communication channels provided by AOS. Internal to AOS, protected communication channels are also used for agent migration.

An important design goal is that AOS should provide mechanisms, but hides its internal implementation from users. In case of secure channel setup and migration, high-level primitives are provided that allow middleware processes to securely authenticate a remote (AOS) process, without having to know about or adopt a specific public key infrastructure or cryptographic toolkit. This mechanism is explained in Section 3.2.2.

3 Architecture of the AOS kernel

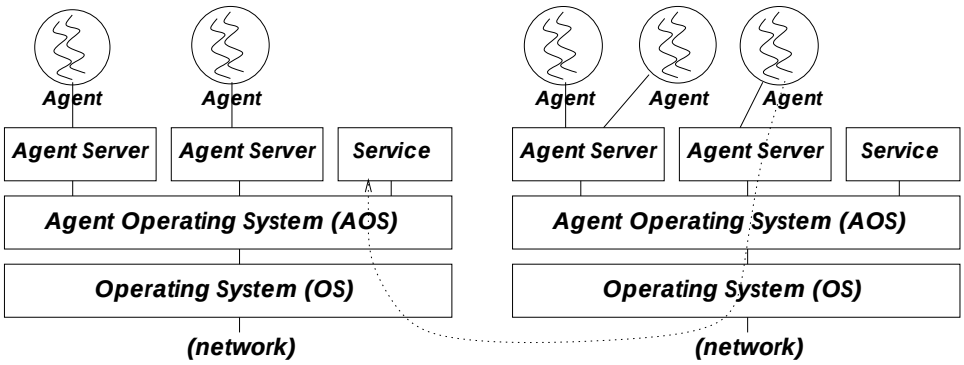
3.1 Architectural model of AOS

AOS intends to provide a 'common base' to a range of specific mobile agent middleware systems. This common base should be viewed as a kernel component in a layered middleware system design. Agent middleware systems can use the AOS kernel for agent code and state management, agent migration and communication, and can extend the AOS layer with middleware-specific components and services, e.g., for agent life-cycle management, middleware management and agent naming services. The architectural model of an agent system using AOS is shown in Figure 1.

AOS provides a means for middleware processes to securely authenticate services and other middleware components in a system, to communicate with these components and services, and to migrate agents to other locations in a secure way. Middleware processes can communicate with each other using socket-like operations over the reliable, ordered and secure communication channels provided by AOS. Multiple communication channels and agent transfer operations from different middleware processes can be multiplexed over a single AOS 'base channel' (Section 3.2.2) for efficiency, e.g., to amortise expensive connection setup times due to cryptographic handshake protocols.

Agent middleware components are distinct processes from an architectural point of view (see Figure 1). The middleware is responsible for providing a runtime environment to agents: AOS is not directly accessible to agents in general. Agents are executed by the agent middleware, which provides them with an API containing middleware-specific primitives.

Figure 1 Example of a layered agent middleware architecture using AOS



Notes: This example system consists of two agent server processes and one service (e.g., a naming service) running on top of AOS on each machine. Mobile agent middleware processes communicate with other local or remote middleware components using AOS. Agents communicate with their runtime environment (e.g., agent server) and do not normally access AOS directly. Example flow of an interaction of an agent with a remote service through the middleware stack is shown (dotted arrow).

AOS comes with a clear specification for interoperability. This specification describes the API available to higher-level middleware processes, including arguments and semantics. The AOS kernel hides differences in the underlying operating system with regard to communication interfaces and file system access from the agent middleware, as middleware systems generally only need to invoke AOS methods to get agent related work done. This increases portability of the middleware system. Agent middleware processes run in a different address space from AOS, and access AOS methods through RPC calls. Besides providing effective fault isolation (Section 2), RPC offers language-independence, as language bindings for different languages can be straightforwardly constructed by generating appropriate client stubs to invoke the RPC calls. AOS supports multiple so-called RPC *dispatchers* for different RPC implementations, which can run simultaneously. Different middleware components can use different RPC interfaces. We currently implemented a binary SunRPC, an XML-RPC, and a Java-RMI dispatcher.

3.2 AOS concepts and primitives

The AOS API provides primitives for agent transport (agent migration) and communication. In addition, AOS provides primitives that allow for protecting resources owned by different middleware components. The agent transport mechanism provides integrity protection of agent code and data, and both the agent migration and the

communication related methods provide a simple yet highly effective authentication mechanism. These concepts and mechanisms are described in detail in this section.

3.2.1 Agent containers

Agent code, data and meta-data (e.g., owner information, time of creation, permissions, etc.) are stored as *segments* (files) in AOS. All segments of an agent are grouped in a data structure called the *agent container (AC)*. The AC is an archive which can contain immutable segments called *persistent segments* (for storing, for example, code) and mutable segments called *transient segments* (for storing, for example, temporary data files). Persistent segments may not be removed after creation, while transient segments may be removed at any time during the agent's itinerary. Each AC has a *Table of Contents (ToC)*. The ToC contains metadata for each segment in the AC, such as creation and modification time, a persistency bit, and a secure checksum (SHA-1 hash) over each segment. The ToC is exposed to higher level middleware processes, which can use this data structure directly or through AOS primitives to find segments (e.g., by name) in the AC. Each segment has a distinct entry in the ToC, indexed by a *SegmentID*. SegmentIDs are used by calls to manipulate segments in the AC.

Before an AC can be shipped to another AOS kernel, it has to be *finalised*. The *finalise* call synchronises any new or changed content of the AC to disk (allowing for crash recovery), updates the checksums in the ToC, and creates a signature over the ToC. When AOS ships an AC, it sends this signature along with the AC. When an AC is received by an AOS kernel, it verifies the ToC (checksums), and the signature over this ToC. A signature over the ToC is also created by the receiving AOS kernel and sent back as a receipt; this receipt can be logged by AOS for auditing purposes, if required. The ToC data structure can be used to implement an efficient multi-hop audit trail verification mechanism that allows for detecting any malicious modifications to an agent's code or data along an agent's migration path (Section 4).

3.2.2 Communication endpoints and authentication

AOS provides a simple socket-like API for communication. Calls include creation and deletion of communication endpoints, connect, accept, send, receive and select calls. These calls allow for setting up and using secure, reliable, ordered communication channels to AOS endpoints.

AOS comes with a simple but highly effective authentication model based on public key cryptography, which is used when connections are set up using AOS. The authentication model is based on the concept of *Self-certifying Identifiers (ScIDs)* (Mazières et al., 1999). A ScID is a SHA-1 hash of the public key of an AOS kernel, where this kernel has access to the associated private key.

Endpoints are created by AOS for AC transport and for communication related purposes. An AOS endpoint is described by an *AOS contact record* that contains the AOS kernel's endpoint information (i.e., IPv4/v6 address and port), and the AOS kernel's ScID. Middleware endpoints relative to AOS are identified by an index field (analogous to a port number) in the AOS contact record. AOS contact records are used by a middleware component to set up a connection or to ship an AC to another agent middleware. As part of connection setup, AOS internally verifies that its peer AOS kernel has the private key corresponding to the ScID in the AOS contact record.

Internally, AOS uses a standard protocol (TLS/SSL) for authentication and key-exchange, to set up an efficient, secure, encrypted channel to the peer AOS. The middleware can specify a cryptographic *cipher suite* for the channel at connection setup time, to influence the strength of the security protocols used by the internal connection. Other than that, the middleware is unaware of the mechanisms used in AOS for secure channel setup. Communication channels and agent shipments over the same pair of AOS kernels, with the same security properties (i.e., cipher suites), are multiplexed over a single AOS 'base channel'. Re-using base channels to multiplex communication channels and do agent shipment operations over, allows for amortising expensive initial secure (SSL) connection setup times.

The advantage of ScIDs over, for example, X.509-based approaches, is that no PKI infrastructure is required to bind keys to names, since ScIDs are coupled directly to keys: a ScID can be used to authenticate an entity directly. How authentication takes place is hidden inside the AOS kernel; the middleware simply specifies a ScID (in an AOS contact record) as part of invoking an AOS operation, or obtains the ScID/contact record of the peer AOS kernel as a result (e.g., when accepting a connection or when receiving an agent). To authenticate a remote AOS kernel, a secure mechanism for passing an AOS contact record suffices. Section 4.1 describes an example of how this can be achieved. Any middleware authentication mechanism (e.g., using a specific PKI) can be layered on top of the AOS abstractions, and systems that do not care about security may even ignore the mechanism if they wish so.

A key property of the AOS authentication model is that the middleware does not have to support a specific PKI or even public key cryptography; middleware processes simply use AOS contact records, and AOS implements the required mechanism for authentication and secure channel setup based on the information available in the contact record. Although conceivably some kind of public key cryptography is required for setting up an end-to-end authenticated channel on top of AOS (see Section 4), AOS does *not* force the use a particular security model or cryptographic implementation upon the middleware system that uses it.

3.2.3 Secure isolation and sharing of resources in AOS

One important benefit of sharing a stand-alone AOS kernel between processes is that it allows for compartmentalisation of the middleware, and for flexible and efficient exchanging and sharing of information stored in AOS between different middleware processes. For example, in an agent middleware system that implements multiple agent server processes for different agent programming languages, instead of moving the content of an AC from a central middleware component to a separate agent server, the only thing that needs to be passed is a credential that allows access to this AC through a shared AOS kernel.

AOS provides a simple but effective authorisation credential called a *cookie*. A cookie is a simple authentication token (basically, a hard-to-guess random bit string), generated by AOS, which is known to the middleware and must be passed with each invocation of an AOS method. Each cookie is associated with a *role bitmap*. The role bitmap specifies which AOS methods may be invoked by the holder of the cookie. For brevity, we refer to a cookie/role-bitmap combination as a *role* for the remainder of this text. A role bitmap must be specified when a new cookie (role) is created by AOS and is

irrevocably connected to this cookie (role). Roles (cookies) can, if required, be passed between processes.

After role creation, its role bitmap is stored in an internal AOS table, together with an (initially empty) list of resources owned (created) by this role. Using roles, AOS can verify whether a method invocation is allowed and whether the resource that is referred to is owned by the invoking role. An AOS resource (e.g., AC or communication endpoint) can only be owned by one role. Child roles (and their subroles and associated resources) are owned and can be deleted by their creating role only. Upon role creation, AOS verifies that the bitmap for the child role does not exceed the creator's role bitmap (permissions).

As an example of using roles, consider a central middleware process which receives incoming agent ACs using AOS, and then dispatches these to an appropriate agent server. The central middleware process creates a new role before it calls an AOS method to receive an AC, as part of the procedure for agent transport (Section 4.4). This role's bitmap only allows AC related calls, and no communication related calls. After receiving an AC, this AC is now owned by, and only accessible to, this role. After inspecting the AC's content and permitting the agent to enter, the central middleware process passes the role's cookie to an appropriate agent server process which can then retrieve the agent's code and data segments directly using AOS calls, and start the agent. An agent server can only access the ACs associated with the roles that it has been given by the central middleware process, and can only invoke operations related to these ACs.

The role model specifically allows construction of modular middleware that adheres to the principle of least privilege. Compartmentalisation avoids that a single compromised middleware component can exceed its privilege (as a role has only the minimum required operations) and prevents compromise of resources owned by different middleware components, such as ACs and communication channels. Different approaches to compartmentalisation are conceivable; the role model is flexible enough to accommodate different compartmentalisation strategies.

4 Building secure mobile agent middleware using AOS

AOS can be used in both open and closed systems; there is no specific usage model embedded in AOS, nor does AOS depend on any central or shared services between all applications. This is illustrated effectively by the fact that AOS is used in the design and implementation of two conceptually very different mobile agent systems, namely AgentScape (an 'open' system) and Mansion (a system that defines closed, application-specific worlds (van 't Noordende et al., 2004)). In effect, AOS provides secure agent transport and communication mechanisms, mediated through an effective access control model that allows its usage in different settings. AOS' minimality ensures that different mobile agent systems can make use of AOS.

As an example of how AOS can be used in a secure way in a concrete system, we describe how AOS is used to implement an Agent Transfer Protocol (ATP) in Mansion. Mansion has a large emphasis on security, which partially influenced the design of AOS. This can be observed from the Mansion ATP discussed in this section.

We start by describing how secure end-to-end communication can be established when using the AOS kernel. Next, some relevant components of Mansion are introduced,

which are required for understanding the Mansion ATP. After that, the ATP is described in detail. Combined, this section demonstrates how a secure agent middleware can be constructed using AOS.

4.1 End-to-end authentication and secure communication

In many cases an end-to-end authentication protocol is required to obtain security at the middleware layer. Mansion uses ScIDs for authenticating middleware processes. To allow for middleware-level end-to-end authentication, the AOS contact record is extended to contain a ScID corresponding to the public key of the middleware. The resulting data structure is called a *Middleware Contact Record (MCR)*. Another middleware may choose a different model. For example, it can include a full (certified) X.509 public key certificate (chain) in its version of the MCR.

Because AOS provides an encrypted transport mechanism, it becomes relatively straightforward to implement a challenge-response based authentication protocol on top of the AOS channel, based on public key cryptography. In such a protocol, a peer middleware process can be sent a challenge using which it can prove that it has access to the private key corresponding to its ScID or certificate. Next, both parties should exchange (authenticated) messages between each other that contain the AOS endpoint information of their own (trusted) AOS kernel. This information can then be compared to information about the peer AOS endpoint and ScID, as verified by their own AOS kernel. If this check does not take place, an impostor AOS kernel may sit between the AOS kernels used by the middleware processes as a man-in-the-middle, which can decrypt and read all information passed over the channel. After AOS endpoint information is exchanged and verified as part of a middleware-level end-to-end authentication protocol, both parties can trust the underlying AOS channel with regard to confidentiality (secrecy), without requiring further cryptography at the middleware level.

4.2 Agent location service

Mansion uses a home-based approach for communication (van 't Noordende et al., 2004). Each agent's homebase consists of an agent location service (ALS), trusted by the agent's owner, which keeps track of the agent's contact information. Each agent has a unique, self-certifying *AgentID*, which contains the ScID of this agent's ALS. Agents use AgentIDs to communicate with other agents. The middleware looks up the agent's current contact address in the agent's ALS (it can find the ALS in a directory service using a special name containing the ALS ScID), in order to establish a communication channel on the agent's behalf.

In Mansion, as in most mobile agent systems, the middleware updates the ALS. However, in most systems there is no way for the ALS to verify update requests, so it is straightforward to change an agent's contact information in illegitimate ways, for example, to mount a denial of service attack against a particular agent.

In Mansion, we solved this problem by having the agent's current middleware (the initial middleware is known to the ALS) start an ALS update transaction, which has to be committed by both the sending and the receiving middleware. This transaction will only be completed after both middleware processes agreed to agent migration. The receiving middleware verifies the incoming agent's AC integrity and possibly some of its content

and commits only if this checks out. Either of the two parties may abort the ALS update transaction at any time. The importance of this mechanism for this paper is that AOS is used to implement agent integrity verification.

4.3 Audit trails

AOS provides basic integrity protection, in that it can verify whether an AC's content corresponds to the ToC with which it was shipped. However, AOS cannot directly inspect what changes have been made to the AC *prior* to the previous AOS kernel (from now on also referred to as a 'hop') that the agent ran on. Because AOS has no knowledge of an agent's specific content, AOS cannot make an informed decision on whether any malicious modifications may have been made to the AC along the migration path that the agent has followed.

Audit trails can be used to facilitate verification of integrity over a multihop itinerary. The original idea of establishing audit trails for mobile agents was described in Karnik and Tripathi (2001) for the Ajanta system. Here, *append-only* containers are used where agents can store data in, and a specific audit trail mechanism is used using which tampering with the append-only container can be detected. Compared to the Ajanta system, AOS ACs are more flexible, as both persistent and transient files can be stored in a single container. Also, the AOS AC is platform-independent whereas the Ajanta solution is Java specific. The AOS ToC was specifically designed such that audit trail construction and verification can be done very efficiently.

An audit trail is established by storing the ToC of an incoming AC – together with the signature over this ToC created by the middleware that shipped it, and the public key of the signer – in a new segment before the AC is finalised and shipped to the next host. By retaining the ToCs of all hops that the agent visited, an audit trail is established using which all changes made to the agent's AC can be traced. Because of the ToC design, it is straightforward to check for illegitimate changes to an AC using a binary comparison algorithm that iterates over all ToCs in the audit trail from first to last.

The Mansion middleware verifies the audit trail at each hop before the AC is accepted and the ALS update committed. This way, an AC that was tampered with can be refused and is effectively contained on the middleware where the illegitimate change was made. The ALS stores a log of the (ScIDs of) all hops that an agent visited, to allow for detection of deletion of part of an audit trail (*rollback*) in case of cycles in the agent's itinerary – for example, when an agent visits a (malicious) host twice.

4.4 Overview of the Mansion ATP

This section gives a detailed overview of the Mansion ATP constructed on top of AOS. The Mansion ATP combines the audit trail verification mechanism and the transaction-based ALS update mechanism explained in the previous section. In particular, agents are only migrated officially by means of an ALS update (where an agent's contact information in the ALS indicates the agent's *official* whereabouts at a particular time) if both the sending and receiving middleware agree on the agent's integrity. Integrity verification combines AOS-level AC integrity verification with middleware-level audit trail verification. Note that the ATP may also be aborted for other reasons than AC integrity violation – e.g., an incoming agent's programming language may not be

supported by the receiving host, or the receiving host may have too few resources available for running the agent.

The Mansion middleware (MMW) waits at an ATP endpoint for incoming requests. The ATP endpoint is a regular communication endpoint, to which other MMW processes can connect using an AOS call (Section 4.1). The middleware makes a preliminary choice on whether it allows receipt of an AC, based on authentication of the peer process and on information embedded in an initial ATP request message.

The ATP protocol is outlined in Figure 2. Below, we give a detailed outline of the protocol including AC transfer and audit trail verification. Performance measurements of most steps in this protocol are given in Section 5. The Mansion ATP, including an ALS update protocol and audit trail verification protocol, consists of the following steps.

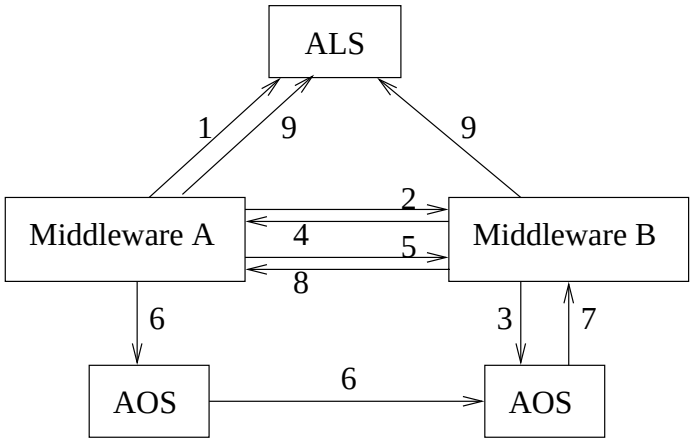
- When a decision is made (typically, by the agent) to migrate an agent, middleware A suspends the agent and all its communication queues, finalises the agent's AC and initiates the ALS update transaction. As part of this, the middleware registers the intended target middleware's ScID in the ALS (1).
- Next, a connection is made to the ATP endpoint of middleware B; middleware A provides information (e.g., programming language, resource requirements) about the agent to the target middleware through an *init message* (2). Based on the init message, the target middleware decides if it is willing to receive the agent.
- If the target middleware is willing to receive the agent, it calls a method on the AOS kernel which creates an endpoint in AOS to which the agent can be sent (3). A unique 'transaction identifier' (XID) is also created by AOS, which has to be used by the sending middleware to ship the AC; this XID prevents that anyone can send an arbitrary AC to an AOS kernel, and enforces that a middleware-level decision to accept an agent precedes shipment/receipt of an AC. The XID, along with the AOS ATP endpoint information, is sent to the client (4).
- The sending middleware signs the ToC of the finalised AC using its own key. The ToC of the finalised AC is obtained by reading out segment 0 of the AC using an AOS method. Middleware A sends the signature over the ToC and its public key to middleware B for verification (5), while simultaneously instructing its AOS kernel to ship the AC (6).
- Middleware B invokes a *wait_ac* call on AOS. This call returns an identifier for the AC after the AC was received and verified correctly (7). An error will be returned if there was any problem with the AC. After receiving a correct AC, its (verified) ToC can be read out by middleware B by reading segment 0 of the agent's AC. Middleware B can now verify that the signature it received from middleware A in Step 5 was made over this AC's ToC.
- If all this checks out, the Mansion middleware searches the AC (using a naming convention) for segments containing earlier ToCs. These segments are numbered sequentially and the Mansion middleware can compare the signed ToCs iteratively using the audit trail verification procedure outlined above.
- If audit trail verification checks out, and if there are no further problems with the content of the incoming AC, the receiving middleware signs the ToC and sends the signature back to the sending middleware as a receipt (8). Middleware A only

commits the migration after it received the receipt from Middleware B. Finally, both parties commit (or possibly abort, in case of problems) the ALS update transaction (9) to complete the migration.

- Before the agent is started, middleware B should store the ToC, key and signature over the ToC created by middleware A, as persistent segments in the AC, to form the next component of the agent's audit trail.

The steps outlined above ensure that agent integrity is verified at all migration steps and that each middleware that sends an agent to another middleware process signs the agent's AC using its private key. In addition, AC integrity verification, ToC signature verification and storage of this information in the AC are required parts of the ALS update protocol, in which both sending and receiving parties must commit the update. This ensures that both middleware processes sign the AC's ToC, that a verifiable receipt can be kept, and that a valid ToC is stored in the agent's audit trail.

Figure 2 Implementation of the Mansion hand-off protocol using AOS



5 Performance of the Mansion ATP protocol using AOS

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 and performance is highly important. In an earlier paper (van 't Noordende et al., 2007b), communication throughput and scalability have been reported for two implementations of the AOS kernel, one in Java and one in C++. These measurements showed good scalability results (van 't Noordende et al., 2007b). We do not repeat these measurements here; instead, we focus on agent transport functionality and present measurements of AOS while performing the ATP outlined in Section 4. We also present some measurements of the latency incurred when communicating over AOS, which complement the measurements described in van 't Noordende et al. (2007b).

The tests were run on a dedicated cluster containing 2.4 GHz dual-CPU/dual-core AMD Opteron DP 280 compute nodes with 4 GB of memory, running a Linux 2.6.18 kernel on an XFS file system using a 1 G Ethernet network, each with a WD Caviar RE,

7200rpm hard-disk with 16 MB cache. The tests were run with a modified C++ AOS kernel that included microsecond timers. The Mansion middleware is written in C and a SunRPC dispatcher was used to invoke methods on AOS. All AOS connections were configured to use 128 bits AES encryption with SHA-1 message authentication.

A test setup was created, consisting of three Mansion middleware processes, each running on a separate node in the cluster. Agents were injected into the Mansion system, transferred to the first middleware and subsequently transferred through two additional middleware processes before being retrieved by its owner. Timing results were taken at each of these nodes. The ALS was configured to use AOS for communication and ran on a different machine than the middleware processes.

We ran tests of the ATP using ACs of three sizes: 500 KB, 1 MB and 5 MB, respectively. Segments in the AC contained 5,120 bytes of random data, with the 500 KB AC containing 100 segments, the 1 MB AC containing 200 segments and the 5 MB AC containing 1,000 segments. The tests were run up to seven times for each AC size. The measurements selected for this paper are median measurements or close to average. We observed some outliers in the ATP tests. Inspection of the middleware log files showed that in these cases, concurrent activity took place in the middleware – for example, a ToC signature was received and verified in the middleware, while AOS was busy unzipping an AC, corresponding to the concurrent Steps 5 and 6 in the Mansion ATP protocol outlined in Subsection 4.4.

Since the Mansion middleware and AOS are concurrently running processes which are multithreaded by design, some interference is inevitable for measurements of the Mansion ATP protocol in a live system. However, since such outliers do not represent pure AOS performance, we chose median values to avoid the effect of those outliers in some cases.

5.1 Middleware to AOS RPC communication overhead

In van 't Noordende et al. (2007b), throughput measurements and scalability have been measured, which show good scalability in that the total throughput remains constant independent of the number of concurrent sends or AC shipments over a single AOS-to-AOS base channel. However, even though scalability is important for a kernel that is to be used by multiple processes concurrently, baseline performance in terms of latency and throughput are probably at least as important for most system designers.

An important constraint with regard to AOS performance is the fact that RPC calls are made to invoke AOS operations. For heavyweight operations, such as an AC transfer, the added overhead is small compared to the overall cost of the (remote) operation and can be mostly neglected. However, for tasks such as communication over an AOS channel, the additional RPC overhead increases latency and decreases throughput. To gain insight in this aspect, we measured the roundtrip time of a single invocation of a local AOS 'ping' method. The measurements include the time it takes the (multithreaded) SunRPC dispatcher to handle the request, verify the cookie and invoke the native 'ping' method, which returns a 32 bit integer. The average roundtrip time of this RPC call is 129 μsec. For comparison, a simple getpid() system call on the same machine takes 7 μsec on average. Roughly speaking, about 122 μsec is added when using an AOS primitive, compared to using an OS primitive (e.g., sockets) directly.

Clearly, because of RPC related overhead, AOS communication is not an optimal solution when low-latency, high-bandwidth communication is required; instead, communication over AOS should be considered primarily useful in cases where there are limits on the number of usable TCP ports, e.g., when a machine resides behind a firewall.

5.2 Finalise costs

Prior to shipping an AC, the AC must be finalised to ensure that the AC's ToC is generated and that all segments are stored safely on disk. The latter is for crash-recovery reasons; finalise acts as a checkpoint of the agent's state. An AC is stored in a zip file internally, to facilitate efficient transport over the network. Finalise constructs a ToC of the AC and signs it, prior to shipping it to another AOS kernel. Finalise syncs the AC to disk for crash recovery reasons.

Table 1 shows a microbenchmark of the finalise costs of agent containers of 500 KB, 1 MB and 5 MB containing random data. ToC checksumming and signing cause little overhead, even for large ACs. Creating a zip file and sync'ing it to disk cause substantial overhead; this can be explained because zipping requires that each segment is copied into the zip file, possibly after compression. Zipping overhead is nearly linear to the total AC size. Sync'ing the resulting zip file to disk is also rather expensive.

Table 1 Breakdown of finalise cost (in milliseconds) for ACs using the C++ kernel

	500 KB	*1 MB*	*5 MB*
Create ToC	8.4	9.1	14.9
Sign ToC	7.7	8.1	11.7
Zip AC	34.2	66.4	321.9
Sync AC	21.7	36.3	102.7
Total	87.6	127.5	450.2

Note: Results for the run with median total cost.

As mobile agents may migrate often during their lifetime, AC finalise and transfer cost can increase the time for an agent to achieve its task considerably and may influence scalability of the mobile agent middleware as a whole. A straightforward optimisation for performance is to have AOS ship segment files to another AOS kernel directly, without zipping the files first, in an FTP-like manner. Note that this could conceivably hurt performance in cases where bandwidth is limited; in such cases, the compression offered by zip files is an advantage. Another straightforward optimisation for performance is to let go of the crash recovery assurance by means of the fsync system call.

5.3 Overhead of the Mansion ATP using AOS

In this section, we describe the performance of the Mansion ATP protocol outlined in Subsection 4.4. In these tests, we measure the ATP overhead after the agent has migrated one hop; thus, the receiving middleware must verify a two-level audit trail.

Table 2 shows the time it takes for the most important steps in the Mansion ATP. We measure the total time it takes to ship an AC of 500 KB, 1 MB or 5 MB consisting of

segments of 5 K containing random data, as well as some of this operation's component costs.

The overall time for the ATP to complete for a 500 KB agent container is 171.9 msec. For a 1 MB AC, the shipping time is 350.6 msec. and for a 5 MB AC this is 726.7 msec. This time does not include the finalise time (taken from Table 1), but does include channel setup, shipment of the zip file, receipt, extraction and verification of the AC and the audit trail at the receiving side, verification of the returned ToC signature and committing the ALS update.

Table 2 Performance of an AOS-based ATP with agent containers of different sizes

Protocol step		*Time (msec)*		
		500 K	*1 MB*	*5 MB*
S	Finalise AC	87.6	127.5	450.2
S	MMW sign ToC	7.7	8.1	9.0
R	AOS extract AC + verify ToC	25.7	215.0	565.6
R	MMW check ToC signature	0.8	1.4	2.5
R	Audit trail verification	1.6	2.6	4.2
S	AOS ship_ac completion	68.4	265.2	636.4
S	MMW-level ATP completion	171.9	350.6	726.7

Note: S and R indicate AC sending, resp. receiving side.

Signing and verifying ToC signatures requires public key cryptography. As can be seen from Table 2, the overhead of these operations, as well as for audit trail verification, is negligible compared to the overall migration cost. The overall migration cost is dominated by zip and unzip times. Unzip times are the major component of the AOS extract AC and verify ToC measurement shown in Table 2. AOS-level AC extraction and ToC verification times are not completely linear with respect to the AC size. The complete AOS ship_ac call only returns if the receiving side has received, extracted and verified the AC. Therefore, the AOS ship_ac completion measurements are dominated by AC extraction cost. The overall MMW-level ATP completion time is somewhat longer than the AOS ship_ac completion time which it includes. This is caused by the additional interactions required at the middleware level, compared to the AOS-internal interactions.

The AOS ToC design was optimised to make efficient (binary) comparison between ToC entries of different ToCs in an audit trail possible. As can be observed from Table 2, audit trail verification poses a negligible overhead compared to the overall overhead: between 1.6 and 4.2 msec. for a 2-level audit trail, depending on AC size; indeed, this is very efficient. Note that for ToC comparison, only access to ToC segments is required, not to other segments. This is because correspondence of the segments in the AC with the ToC entries of these segments has already been verified by AOS (wait_ac).

In all, Table 2 shows that the Mansion ATP can be implemented with little overhead compared to the basic cost for finalising an AC and transferring it to another AOS kernel over a secure AOS channel. The major cost component is zipping and unzipping the AC, as well as sync'ing the AC to disk. ToC and audit trail verification mechanisms are very efficient and cause negligible overhead, even though a 5M AC consists of a very large number of segments.

6 Related work

AOS has a design which is not directly comparable to existing work. AOS is not an agent middleware itself, but rather a middleware building block. In this section, we describe related work that is partially related to AOS or the AOS requirements.

The FIPA (http://www.fipa.org) standard specification includes a series of documents describing the functionality and operation of agent middleware. FIPA compliant agent middleware can interoperate with 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 (Bellifemine et al., 2001). 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 are considered application specific and can be arbitrary location or yellow pages services such as DNS or LDAP servers.

Ajanta (Karnik and Tripathi, 2001) is designed to include a number of security primitives and architectural 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 van 't Noordende et al. (2004). However, Ajanta is completely Java-based and is not designed to incorporate or interact with other software components or services, while AOS is platform (middleware) and language-independent.

The Tacoma (Johansen et al., 1995) project focuses on operating system support for mobile agents. In that respect, it has many similar design goals as AOS by providing low-level 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 (Gutknecht and Ferber, 2000) 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 architectural model with AOS, the design and implementation are 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', realising different execution environments for running an application. Alternatively, AOS aims to provide a minimal, secure middleware layer for constructing mobile agent systems, and is not directly used by agents.

7 Discussion

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 middleware 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. A simple but effective access control mechanism ensures that different middleware processes cannot access or compromize AOS resources of other middleware processes.

AOS provides a minimal set of primitives that are common to mobile agent systems, in particular for agent code and data storage, agent transport, and communication between 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, secure agent transport, and mobile agent audit trails.

AOS offers a flexible basis for the construction of secure mobile agent systems and for deploying multiple services or middleware processes at the same time on a single AOS kernel. Support for secure middleware that *internally* consists of components written in different languages is a novel contribution of our work. The access control model based on roles allows for applying the principle of least privilege within a modularly designed agent system and offers separation of resources in scenario's where AOS is shared between different mobile agent systems or middleware components.

Two implementations of AOS (in Java and C++) have been built, used, and tested for interoperability. Scalability measurements of the AOS in an earlier paper showed that AOS scales well with concurrent use. This paper shows that it is feasible to build a secure mobile agent transfer protocol on top of the abstractions provided by AOS. Measurements show that performance of the Mansion ATP is dominated by disk I/O related costs, rather than by security related costs. Measurements show that Mansion's audit trail verification mechanism can be implemented on top of AOS very efficiently. AOS is used in two different agent systems designed in our department, which illustrates that AOS provides the right level of abstraction to construct diverse mobile agent systems – even if these systems have rather different requirements or designs.

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

Baumann, J., Hohl, F., Strasser, M. and Rothermel, K. (1997) *Mole – Concepts of a Mobile Agent System*, Technical Report, August, Universität Stuttgart.

Bellifemine, F., Poggi, A. and Rimassa, G. (2001) 'Developing multi-agent systems with a FIPA-compliant agent framework', *Software – Practice and Experience*, Vol. 31, No. 2, pp.103–128.

Gray, R.S., Kotz, D., Cybenko, G. and Rus, D. (1998) 'D'agents: security in a multiple-language, mobile-agent system', *Mobile Agents and Security*, pp.154–187, LNCS 1419, Springer-Verlag.

Gutknecht, O. and Ferber, J. (2000) 'The MADKIT agent platform architecture', in *Proceedings of the International Workshop on Infrastructure for Multi-Agent Systems*, June, Montreal, Canada, pp.48–55.

Johansen, D., van Renesse, R. and Schneider, F. (1995) 'Operating systems support for mobile agents', in *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, May, Orcas Island, WA, pp.42–45.

Karnik, N. and Tripathi, A. (2001) 'Security in the Ajanta mobile agent system', *Software – Practice and Experience*, April, Vol. 31, No. 4, pp.301–329.

Mazières, D., Kaminsky, M., Kaashoek, M. and Witchel, E. (1999) 'Separating key management from file system security', in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp.124–139.

Milojicic, D., Douglis, F. and Wheeler, R. (Eds.) (1999) *Mobility: Processes, Computers and Agents*, ACM Press.

Suri, N., Bradshaw, J.M., Breedy, M.R., Groth, P.T., Hill, G.A., Jeffers, R., Mitrovich, T.S., Pouliot, B.R. and Smith, D.S. (2000) 'NOMADS: toward a strong and safe mobile agent system', in *Proceedings of the Fourth International Conference on Autonomous Agents*, June, Barcelona, Spain, pp.163–164.

Tripathi, A.R., Ahmed, T. and Karnik, N.M. (2001) 'Experiences and future challenges in mobile agent programming', *Microprocessor and Microsystems*, April, Vol. 25, No. 2, pp.121–129.

van 't Noordende, G., Balogh, A., Hofman, R., Brazier, F. and Tanenbaum, A. (2007a) 'A secure jailing system for confining untrusted applications', *International Conference on Security and Cryptography (SECRYPT)*, 28–31 July, Barcelona, Spain.

van 't Noordende, G.J., Brazier, F.M. and Tanenbaum, A.S. (2004) 'Security in a mobile agent system', in *Proceedings of the First IEEE Symposium on Multi-Agent Security and Survivability*, August, Philadelphia, PA, pp.35–45.

van 't Noordende, G.J., Overeinder, B.J., Timmer, R.J., Brazier, F.M. and Tanenbaum, A.S. (2007b) 'A common base for building secure mobile agent middleware systems', *Proc. 2nd Int'l Multiconference on Computer Science and Information Technology (IMCSIT)*, October, Wisla, Poland, pp.13–25.

White, J.E. (1996) 'Telescript technology: mobile agents', White paper, General Magic.

Wijngaards, N.J.E., Overeinder, B.J., van Steen, M. and Brazier, F.M.T. (2002) 'Supporting internet-scale multi-agent systems', *Data and Knowledge Engineering*, Vol. 41, Nos. 2–3, pp.229–245.

Notes

1 The AOS specification can be requested from the authors.