An Efficient Implementation of the Agent Operating System
Masters Thesis

R.J. Timmer
Vrije Universiteit, Amsterdam

August 29, 2005

Supervisors:	prof.dr.	Andrew S. Tanenbaum
	drs.	Guido J. van 't Noordende
Second reader:	prof.dr.	Frances M.T. Brazier

Abstract

This MSc thesis presents the architectural design and evaluation of the C/C++ implementation of the Agent Operating System (AOS). This C/C++ implementation has been designed to interoperate with the existing Java implementation, which means that the C/C++ AOS kernel implementation (1) is capable of setting up a connection to a Java implementation, transferring an agent to the AOS Java kernel, and receiving an agent from the AOS Java kernel (2) is capable of supporting communication between agents on different kernels.

To this purpose, the C/C++ implementation implements the full version 5 (rc3) AOS API, which includes encrypted and multiplexed communication channels, storage and transfer of agent code and data and role management. The implementation is multi-threaded and uses a SunRPC dispatcher to contact the AOS kernel instance.

Performance tests have been done to inspect the differences in performance between both implementations.

Preface

This thesis contains the results of a MSc project, completing a six year period of studying Computer Science at the Vrije Universiteit, Amsterdam. The project that was the base for this thesis occupied most of my time for the entire last year of the study. Still, this could not have been possible without the contributions of various people.

First and foremost, I would like to thank Guido van 't Noordende. Not just for for supplying me with a fun project but as well for all the support during the entire project. I would never have thought that I could do a project that covered so many of the subjects of the entire Computer Systems systems that I really liked, in that sense the subject of the project was perfect for me. Also thanks for your numerous comments and suggestions on the final thesis.

Also, I would like to thank all the other people that have given their opinion and (many) suggestions for this thesis, the result would not have been the same without the contributions of Andy Tanenbaum, Frances Brazier, Michel Oey, Benno Overeinder and of course Guido van 't Noordende.

This thesis describes the result of implementing the Agent Operating System (AOS). Although I tried to find how to implement most of the details from the AOS specification, some help was also received by the following people during programming and testing. Thanks to Rutger Hofman for suggestions and helping me to find out problems with the C/C++ AOS kernel implementation. Also, thanks to Arno Bakker and Michel Oey for support on the Java AOS kernel implementation.

Table of Contents

Preface....................3
1 Introduction....................6
1.1 Agent Systems....................6
1.2 Related Work....................8
1.3 AOS Project Goals....................9
1.4 Outline....................10
2 AOS Goals, Requirements and Design....................11
2.1 AOS Goals and Requirements....................11
2.2 AOS Functional Design....................13
2.3 Architectural Design Implications....................13
2.4 Introduction to the AOS API....................15
2.4.1 Connections....................16
2.4.2 Agent Containers....................17
2.4.3 Role management....................18
2.5 MSc Project Goals....................19
3 AOS Software Architecture, Design and Implementation....................20
3.1 AOS Implementation Architecture....................20
3.2 Dispatcher....................22
3.3 Cookie and Role Management....................23
3.3.1 Cookie Table....................25
3.4 Agent Container Management Module....................26
3.4.1 AC Finalizing....................27
3.4.2 Segment Caching....................29
3.5 Connection Management....................32
3.5.1 Base Connection Architecture....................33
3.5.2 Virtual Connection Setup....................34
3.5.3 Base Connection Selection....................36
3.5.4 Virtual Connection Administration....................38
3.5.5 Local Communication....................39
3.5.6 Flow Control of MUX & ACTP....................40
3.6 Removing Unused Resources....................41
3.7 Cookie Crash Persistence....................42
4 Results & Testing....................44
4.1 Test Setup....................44
4.2 Performance Tests....................45
4.2.1 Dispatcher Performance....................45
4.2.2 Virtual Connection Performance....................48
4.2.3 AC Finalize Performance....................53
4.3 Correctness Testing....................59
5 Conclusion....................61
6 Future Work....................62

Appendix A: required technologies....64
A.1 External Data Representation (XDR)....64
A.2 SunRPC....65
A.3 OpenSSL....65
A.4 Pthreads....66
A.2.1 Mutual Exclusion using Mutexes....67
A.2.2 Condition Variables....68
A.5 Zip data compression....68
Appendix B: Programming on AOS....69
B.1 AOS Code Organization....69
B.2 AOS data structures & thread-safety....70
B.2.1 Concurrent Data Access with Blocking Function Calls....74
B.3 Dispatcher Implementation....74
B.4 OpenSSL Cipher Suite Negotiation....76
Appendix C: Source Code Listings....78
References....81

1 Introduction

This document is intended to accompany the C/C++ implementation of the Agent Operating System (AOS). Together they are the result of a MSc project included in the Computer Science MSc program at the Vrije Universiteit Amsterdam, based on the AOS specification version 5 rc3 [1]. This document provides an overview of the entire project and results.

Chapter 1 gives a general introduction to agent systems, after which chapter 2 presents the most fundamental aspects of the AOS design and specification. In chapter 3 the focus is on the software design, architecture and implementation of the AOS kernel that was made for this project. Chapter 4 describes the testing that was done on the C/C++ kernel implementation. Chapter 5 contains a discussion of the entire project, after which chapter 6 highlights future work that can still be done on the AOS implementation and specification.

Three Appendices are supplied. Appendix A explains related technologies which were used for the AOS implementation. Appendix B is an introduction suitable for anyone who intends to modify the source code of the AOS implementation. Appendix C contains the most relevant source code listings.

1.1 Agent Systems

Agents are individual software entities that usually work in a distributed environment. The term *agent* covers many different types of agents; its exact meaning usually depends on the context of the entire agent system for using these software entities. Different classifications can be made between various types of agents [14].

There are some common properties that apply to most agent systems, and usually many differences. Typically, an agent itself acts autonomously to achieve a certain goal, which it does on behalf of its owner. Agents can contain some level of intelligence (either fixed rules or more advanced forms of artificial intelligence) to help them in performing their tasks. Agents can be aware of their environment, and able to reactively or proactively react to the state (and state changes) of this environment. Agents can also have social ability, meaning they can interact with the user, the system they work on or with other agents.

A small distinction that can be made is between *multi-agent* systems and *mobile agent* systems, although there is not a strict difference between both. In multi-agent systems, multiple agents typically cooperate to accomplish a common goal. Agents can be stationary or mobile. Mobile agents usually travel between various host machines to accomplish their own (individual) goals. Mobile agents that move between individual host machines in a system to gain local access to the resources on these machines. Often, an agent process itself decides when (and where) it wants to be relocated. Because mobile agents are software processes, relocation of an agent means that the running process has to be moved. This process of relocation is often referred to as *code migration.* In short this means that the running process has to be suspended, (some of) its state has to be transported and the process needs to be restarted on the remote machine.

The focus of this thesis is mainly on agent platforms that support mobility. There are two distinct types of agent and code migration. There are mobile agent platforms that need to restart an agent process from the beginning after the agent is moved (*weak migration*), and

systems that can restart an agent process such that it can continue execution where it left off (*strong migration*). Migration not only implies that an agent program has to be migrated, but also some (runtime) state of the agent process has to be transferred. If the mobile agent system offers strong migration, usually the platform itself can take care of automatically transporting the data and state belonging to the agent (a process often referred to as serialization). If only weak mobility is offered, an agent must manually serialize its own contents. Strong mobility usually implies some limitations on the programming language that can be used for agents, while weakly mobile platforms might offer more freedom in the choice of languages for implementing mobile agents. In particular, standard (already existing) programs might even be used in weak mobile platforms. Weak mobility generally requires no extra operating system support for moving an agent process state, as would be the case for binary agents for strong migration, in which case at least the program stack and registers have to be moved. So a platform offering weak mobility is generally easier to implement, and it usually offers more flexibility for the agent (such as choice of programming language).

Agents can generally not work directly on top of a standard operating system or the network because of the lack of serialization and migration primitives, which is why usually there is at least one extra layer on top of the operating system with which the agents can interact (this is often called an *agent middleware system* or an *agent platform*). Agents can contain potentially sensitive information, so it is important that an agent middleware system can guarantee some level of secrecy of the agent contents (so no one is able to extract private information from the agent) and some kind of integrity protection for the agents (so that tampering with the contents can at least be noticed by the agent owner).

Because agents can contain arbitrary program code, protection measures also have to be offered for the middleware systems themselves. Agents must be restricted in the actions they can perform on the system they are running on. As a bare minimum, agents must never be able to access resources (e.g. files, databases) they are not allowed to access, and usually there should be a limit present on the amount of the resources that an agent is allowed to consume (for example CPU cycles).

Mobile agent systems are usually based on the idea that moving agent program code around a network is generally cheaper than transporting large amounts of data over a network. This is often useful for agents that need to do large data analysis: these agents can be sent to a remote location where they can work on the local data. Mobile code in this case is useful for reduction of the network load, especially when network bandwidth is too limited for performing the data analysis using traditional (client-server) approaches. Another advantage of mobile agents is when the owner is often disconnected from the network, so the owner can send out the agent to perform its task, and collect the results later when the agent returns. Interaction with resources located on remote machines may also be required if the resource owner does not want to have to transport the data over the network, for example a museum database. In this case it is better to send out an agent to inspect the data locally, after which it can reply to its owner if the host has something of interest.

1.2 Related Work

Mobile agent systems (MASs) are not new: many systems and platforms have appeared since Telescript [6], which was the first commercial mobile agent system. Mobile agent systems consist of pieces of executable code that can be migrated over different machines in a network. The concept of this mobile *code* is also not entirely new: even PostScript (which is mainly used to transport jobs to a printer) is essentially a program written in PostScript to be transported to a remote location, usually to give instructions to a printing device. For a good background on mobile code and some existing systems see [2].

Many different MASs have been developed over the years, all with their own goals in mind. Some focus heavily on the support for strong migration, which was realized in Telescript by offering a dedicated language for the mobile parts of the application. The rest of the application can be written in any language however, so that the static and mobile parts of the application could interact. The Telescript language was specifically created with strong migration in mind. A Telescript program runs on a Telescript interpreter, much like the (more recent) Java virtual machine. Telescript was a commercial (and to some extent unsuccessful) product and it was eventually withdrawn from the market. However, it offered a lot of inspiration for the developers of other mobile agent systems.

More recent systems that offer strong mobility usually require a high level programming language. Both the D'Agents [7] system (which supports TCL, Java and Scheme) and the ARA [8] system (which supports agents programmed in Java, TCL and C/C++) use dedicated interpreters for each different language. These systems are both layered: different components are present for handling programming language specific aspects (language interpreters) and for the generic functionality (like the agent environment, migration or security). The level of mobility that is offered in D'Agents depends on the programming language that is used: D'Agents supports weak mobility for agents programmed in Scheme, and strong mobility for agents that are using TCL or Java. ARA offers strong migration for all languages, and it does so by offering interpreters for all of the languages that it supports. This kind of structuring offers some flexibility and extensibility.

Many of the newer agent systems seem to rely (solely) on the Java language (and platform) as a base for mobile- or multi-agent systems. Examples of these are the (older) IBM Aglets [9], Ajanta [10] and Jade [11]. Java has two very large advantages, which are its widespread use and its platform-independence, so Java code is already portable. Having portable code does not necessarily mean that the code is mobile: the Java virtual machine does not support migration of the runtime state of a running program, which effectively prevents any of these systems from providing strong migration without using special measures (e.g. modifying the Java Virtual Machine). Moreover, there are inherent security drawbacks when running multiple agents (usually these run as separate threads) in a single virtual machine.

There are a lot of vulnerable spots in Java based systems: if Java security is not configured well, then it can become very easy for an agent to do illegal data access, crash the virtual machine or more. An example of these vulnerabilities is that a Java program can supply custom methods for the deserialization of the program data. The custom routines for deserializing the program code are executed by the server thread that accepts the agent, so that the agent can already use the server thread of execution before the agent itself is even completely accepted. Similar tricks exist for supplying finalizer methods, which are called by the garbage collector thread of the virtual machine once an agent can be cleaned up. If a

finalizer method never finishes, then the garbage collector can never collect any more memory and the virtual machine will eventually choke due to a lack of memory. For more Java problems see [15].

There are currently two mobile agent systems that use AOS as a base, which are AgentScape [13] and Mansion [12], which are both designed to support agents written in different programming languages. Currently, AgentScape supports agents written in Java and Python (support for more languages can be added later) and a weak migration model. Mansion has an emphasis on security and a weak migration model and it specifically aims at supporting agents written in any kind of programming language. In Mansion, agents can run directly on top of the operating system, but for protection reasons, these agents are executed in a safe context (a jail) to limit the access to the system that the agent has.

Not all of these mobile agent systems are designed with the same goals in mind. The Java-only systems can relatively easy implement agent transfer to any kind of machine, in particular if only weak mobility is supported. Strong mobility is a lot harder to achieve and usually requires the use of a modified virtual machine. Systems that do support strong mobility must often rely on some other high-level languages.

Security is also one of the more important aspects of mobile agent systems. If integrity and security for the agent and its environment cannot be guaranteed, then it will be hard to use in a commercial setting, for example when having an agent carry out tasks that use actual money transfers. The possibility of creating real applications also depends on the scalability of the system as a whole: if the entire system is too limited in its actual scalability then its practical use will be quite limited. Both AgentScape and Mansion address these aspects.

1.3 AOS Project Goals

Many agent systems share common basic properties, for example their migration support, their security requirements etc. The Agent Operating System (AOS) was designed starting from the recognition that both the Mansion and AgentScape mobile agent systems have many of these properties in common. AOS provides some of the core concepts of both these systems, so it serves as a base for implementing a (mobile) agent system.

Currently, Mansion and AgentScape use the same AOS, but AOS is not necessarily limited to these platforms, nor is it limited to systems that have a similar migration model (which is weak in both Mansion and AgentScape) or systems with similar security requirements. The functionality is generic: AOS does not force a system to use a specific kind of design philosophy before it can use AOS as a base for implementation.

AOS is aimed at supporting large-scale agent systems. Different kinds of agent systems should have the possibility to work together using AOS, independent of the language in which a system or its agents are implemented. AOS should at least be able to fill the most common needs of all agent systems, while still offering a set of important functions as security, communication and agent transfer.

1.4 Outline

This thesis provides an introduction to the entire philosophy, design and key elements of the Agent Operating System in chapter 2. For anyone unfamiliar with AOS and/or its design goals this is the place to start.

In chapter 3, the software architecture and design of the AOS implementation in C/C++ that was made for this MSc project are discussed. Some of the more technical details of both the specification as well as the implementation are given in this chapter

Chapter 4 provides details about the performance testing that was done for this implementation, as well as a comparison with the Java implementation of the AOS kernel.

A discussion about the project is given in chapter 5, after which chapter 6 presents a list of future work, including possible additions to the C/C++ AOS implementation, possible extra tests that can be done to further analyze the performance of the AOS implementations, and possible additions to the AOS specification that could be useful to a version 6 of the AOS API.

Three appendices are supplied. For people who are unfamiliar with any of the required technologies (SunRPC, XDR, etc) it may be good to have a look at the information in Appendix A, which quickly introduces the related technologies and how these are needed and used in AOS.

For those that want to make modifications to AOS, Appendix B contains some of the issues and solutions that were encountered during the AOS development, and this appendix contains explanations and motivations for the solutions that were used in the AOS implementation.

Appendix C contains some relevant source code listings, also targeted at AOS programmers, of the standard C++ container classes that were used throughout the entire AOS implementation to store and retrieve the AOS data structures in a way that prevents race conditions between threads.

2 AOS Goals, Requirements and Design

The entire AOS project was started from the viewpoint that the Mansion and AgentScape mobile agent systems have a great deal of functionality in common. This is not just the case for these two systems, but many other mobile agent systems also have common needs for offering communication, agent transport and security. AOS provides a set of services which can be used as a base for implementing (mobile) agent systems. Both Mansion and AgentScape build on AOS to implement their specific design and goals.

First we will look at the functionality AOS actually needs to offer in order to provide the needs of different high level agent systems in section 2.1. After that the AOS functional components that are provided are highlighted in section 2.2. Section 2.3 explains the implications of these choices for the structuring and architecture of AOS itself, including the design decisions that were made to meet these architectural constraints. Section 2.4 gives a small overview of the current AOS specification (version 5), after which 2.5 gives an overview of the C/C++ implementation of the AOS kernel.

2.1 AOS Goals and Requirements

The AOS specification [1] can be used as a guide to make an implementation of AOS. Each implementation based on this specification should be able to cooperate with other AOS implementations based on the same specification, though they may be written in different programming languages. Any AOS implementation should provide the same interface to agent middleware systems independent of the platform on which it runs. AOS should provide portability as far as AOS interactions are concerned. In a sense, AOS is a middleware for agent middleware systems. Agent middleware (services) using AOS may be written in different languages, even different processes of the same middleware or application should be able to use the AOS functionality at the same time.

Each individual agent middleware system may however have completely different approaches and requirements as far as the main functionality of the agent middleware itself is concerned. Agent middleware systems that have high demands for security, speed or scalability of the complete system are not very likely to use AOS as a base if it falls far short on one of these properties. On the other hand, adding a lot of functionality to AOS to support (for example security and trust management) might satisfy the needs for very secure agent middleware systems, but this might also force developers of other systems towards this direction, even if these systems do not need to be bothered with security at all. Similarly, offering a strong migration model might satisfy the needs for systems that need this, but when this forces the use of a specific programming language then AOS is not likely to be usable for many other agent middleware systems.

So, offering too much functionality might make the system too heavyweight, or too difficult to use, whereas providing too little functionality would not motivate its use for many systems. Therefore there are some requirements listed here with which AOS was designed in mind. These requirements are:

- AOS should be *programming language independent*: agent middleware systems that want to use AOS should not be restricted to a particular programming language, but middleware systems should be able to use AOS without having to use a special programming language. AOS should also be able to interoperate with other AOS implementations, even though they can both be running on different platforms or implemented in different programming languages. Also, agents running on those middleware systems (implicitly using AOS) may be written in any language.
- AOS should be *minimal*: it should provide a small set of basic functionality which is common to all agent systems. The components that AOS offers can be used to build any kind of agent middleware system. AOS should not provide too much, as this could potentially force a middleware system to use a certain approach to, for example, using security or agent migration. Such design decisions are best left open to the middleware designer.
- AOS should be reasonably *efficient*: a system that provides only a small amount of functionality should really be lightweight. AOS should not introduce extra overhead, which can also help to realize a more scalable system. This means that AOS should provide local services only, and exhibit or implement no complex distributed properties like replication.
- AOS should be *secure*: all primitives that AOS offers should be secure, or at least configurable for security, so that whenever an agent middleware wants to use these primitives in a secure way, it should at least have the possibility to do so. AOS should not force any security policy on top of this, as a middleware can have completely different needs for security, or it can even have no security requirements at all.

These requirements explicitly do not include an agent execution model. The main reason for not managing processes in AOS is that different agent middleware systems can have different requirements and approaches to running agents: some may use a separate process for running each agent, whereas other middleware systems use a threading model, in which case the agent can be inserted into a process or (virtual) machine in a completely different way. These differences make it almost impossible for AOS to do agent process management (or a migration model) that is suitable for nearly all agent middleware systems, so this is completely left out of the requirements for AOS (the middleware using AOS must take care of that).

In conclusion, AOS should offer this basic set of features that most agent systems should be able to use for their own functionality. Agent middleware systems can either extend the functionality if they desire to do so (e.g. by extending the security AOS offers to implement a stronger trust and/or security model), or simply use these AOS features without building any extras on top of this. With the requirements listed here, AOS should provide the necessary basic elements for different parts of an agent system, which could be anything like, for example, an agent server, object server or an agent location server.

2.2 AOS Functional Design

The functionality that AOS should offer can be roughly divided into two categories: providing a communication interface and allowing secure agent storage and transfer.

For communication, AOS offers reliable communication channels, with a configurable level of security. Depending on the requirements of the agent middleware, AOS can use highly secure connections, only moderately security or hardly any security at all. A middleware system can use AOS to create a communication channel to a (remote) middleware system if it also uses AOS, over which inter-agent communication messages, or middleware-to-middleware messages can be exchanged.

AOS only offers a generic (language independent) agent storage space: for each running agent, the middleware can create a different storage space, called an *Agent Container (AC)*. An AC essentially provides a dedicated portable file system for each agent to store its data, and possibly for the middleware to store data about the agent. A similar approach is taken in the Ajanta system [10]; however, the Ajanta mechanism is Java specific and less generic.

To support the migration of agents, AOS offers mechanism to transfer these containers to remote locations, where the middleware itself can use the contents of an AC to take care of restarting the agent after AOS transported the container, so AOS can support systems using both a weak or strong migration model. AOS offers some basic security mechanisms to to guarantee the integrity and secrecy of the data stored in the AC during transfer. Middleware systems can build a more extensive security or trust model on top of this, as for example is done in Mansion [12].

Different middleware systems or programs should be able to use the AOS functionality at the same time. Each one of these middleware systems might be allowed to allowed to use all of the AOS functionality or only a subset of the AOS functionality. The concept of a *role* is used to distinguish the different allowed functionality of entities that are allowed to use AOS: the role of a middleware can prevent access to (specific parts of) AOS. This is especially useful if the middleware consists of different components, which should not all be able to use the same part of the AOS functions so that, for example, an object server can be prevented from using the Agent Container functions. Using roles is also a way to make sure that all the different (parts of) middleware systems are unable to interfere, so different running systems will not be able to access each other's AOS resources (e.g. connections or ACs).

2.3 Architectural Design Implications

Although AOS does not really force a certain design model on the middleware systems, there are some implications of the AOS requirements for its own model. The most important design decision that was made for the AOS system is to have AOS running as a separate process instead of implementing AOS in a (shared) library for example. This allows multiple middleware systems or processes to use the functionality of the same AOS process at the same time, even if they are written in different languages. This layered design is shown in the next figure.

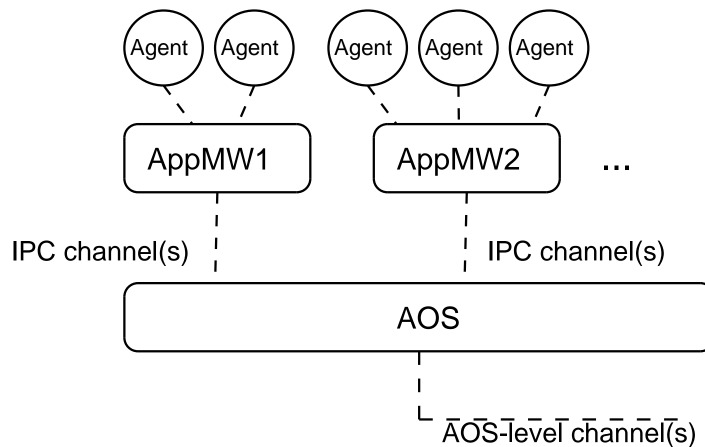

Figure 1: AOS placement

Figure 1 illustrates the placement of AOS with respect to the upper layers: at least one AOS process runs on a host, and different middleware system processes (which might be agent servers, object servers or anything else) can use this AOS process. This means that there is to be an inter-process communication (IPC) channel offered by AOS. For security reasons, middleware processes can currently only use a local communication channel to interact with AOS. AOS is a single process, but more than one AOS process can be running on the same host, so that different users can run their own AOS process on the same host.

Middleware systems are often called *Application Middleware* (AppMW) systems throughout the AOS specification. Different kinds of AppMW systems can be run on top of AOS and all of them can use the AOS primitives. Multiple middleware systems can simultaneously run (as separate processes) on top of a single AOS. More than one AOS instance can be running on a single host, and different AOS systems do not interfere because they use different communication channels. Middleware layers can use different roles for communicating with the AOS kernel, so these middleware layers will also not interfere with each other. Communication between various middleware systems can go by means of the communication channels that AOS offers. AppMW processes can use AOS to contact other middleware process running on the same or on a different (possibly remote) AOS. Usually, some agents (and services) run on top of the AppMW level, but this depends entirely on the structure of these layers themselves: AOS neither prescribes nor assumes anything about the organization of these middleware layers.

A middleware process can contact AOS over the IPC channel using some remote procedure call (RPC) protocol, after which internally to AOS the RPC request is translated into a real function call onto the native AOS code. After executing that function, the result is sent back over IPC. However, the specific RPC protocol, and how IPC requests are transported is left to a specific part of the AOS kernel called a *dispatcher*. This dispatcher is the main entry point for function calls from middleware processes. Different dispatcher types and implementations can be added to the AOS kernel, and more than one can be active at a time. This allows AOS the be accessed by middleware systems written in different languages, using various kinds of transports, like SunRPC over Unix domain sockets, or XML-RPC over a TCP connection. This dispatcher structure is shown in figure 2.

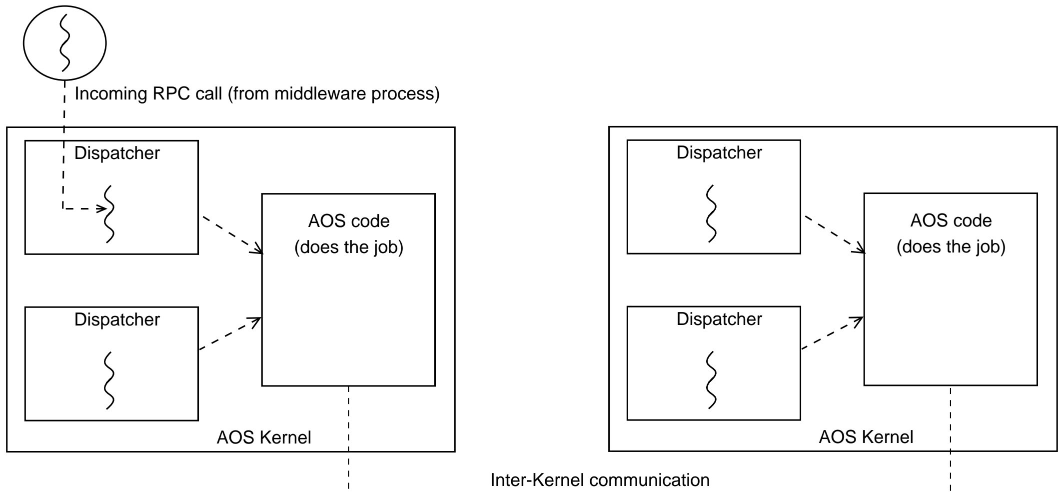

Figure 2: AOS dispatcher structure

Figure 2 shows two AOS kernels, these can be on the same or on a different machine. Both AOS instances have two different dispatchers running. Each dispatcher has a dedicated thread waiting for incoming requests, and each dispatcher can use different RPC languages and transports (such as sockets, pipes, shared memory). Once a dispatcher receives a request, it decides what AOS function call must be executed. Once the function is executed by the native AOS code the result can be sent back over the IPC channel. How this dispatcher thread executes this call is left completely open to the AOS and dispatcher implementations.

2.4 Introduction to the AOS API

The AOS specification presents a complete list of function calls (the AOS API). For reference, this list of functions is included in figure 3. Even though each dispatcher can use its own RPC language to offer access to AOS, the interface offered by each dispatcher should be at least functionally equivalent to the native AOS interface.

ID	Function name
1	int **create_role** (cookie_t parent, role_bitmap_t rb, cookie_t* child)
2	int **delete_role** (cookie_t parent, cookie_t child)
3	int **create_ac** (cookie_t c)
4	int **prepare_wait_ac** (cookie_t c, char sec_suites[ ][ ], struct endpoint* actp_endpoint, xid_t *xid)
5	int **wait_ac** (cookie_t c, xid_t xidlist[ ], int listsz, struct endpoint* src_endpoint, xid_t* xid, int block_time)
6	int **ship_ac** (cookie_t c, int acid, struct endpoint* actp_endp, char sec_suites[ ][ ], xid_t xid)
7	int **delete_ac** (cookie_t c, int acid)

ID	Function name
8	int **read_toc** (cookie_t c, int acid, int offset, int n, struct tocent t[ ])
9	int **create_seg** (cookie_t c, int acid, segtype_t type, char subtype[ ],char description[ ])
10	int **read_seg** (cookie_t c, int acid, int segid, int offset, int n, char tobuf[ ])
11	int **write_seg** (cookie_t c, int acid, int segid, int offset, int n, char frombuf[ ])
12	int **delete_seg** (cookie_t c, int acid, int segid)
13	int **make_persistent** (cookie_t c, int acid, int segid)
14	int **finalize_ac** (cookie_t c, int acid)
15	int **create_listen_endpoint** (cookie_t c, unsigned int required_index, char sec_suites[ ][ ], struct endpoint* the_endp)
16	int **delete_listen_endpoint** (cookie_t c, int endp_descr)
17	int **accept** (cookie_t c, int endp_descr, struct endpoint* from_endp, bool_t block)
18	int **connect** (cookie_t c, struct endpoint* target, char sec_suites[ ][ ])
19	int **send** (cookie_t c, int conn_id, int len, char buf[ ])
20	int **recv** (cookie_t c, int conn_id, int len, char buf[ ], bool_t block)
21	int **peek** (cookie_t c, int conn_id, int len, char buf[ ], bool_t block)
22	int **close** (cookie_t c, int conn_id)
23	int **select** (cookie_t c, int setsiz, int read_dsc[ ], int write_dsc[ ], int except_dsc[ ], bool_t block)
24	int **get_parm** (cookie_t c, char parameter_name[ ])
25	int **reenable_role** (cookie_t c)

Figure 3: Specification of the AOS interface

The AOS functionality can be roughly divided in three categories: role management (functions 1, 2 and 25), Agent Container management (functions 3-14) and connection management (functions 15-23). The get_parm call can be used to query AOS for extra information but is not really relevant for this discussion. The AOS specification goes into detail about what these functions do. This section will give a short overview of the functions AOS offers. For more details see the AOS specification [1].

2.4.1 Connections

AOS allows middleware processes to communicate with each other. The interface to these connections (functions 15-23) is quite similar to communicating with regular UNIX/TCP sockets.

AOS offers the connect call, which can be used by a middleware process to connect to any other middleware process that established a communication endpoint on some AOS,. This may be a process running on the same local AOS, but it may also be running on some other

AOS. A remote process must create a *listen endpoint* on which it can call accept to receive incoming connections, and on which remote processes can connect to. After this, read, peek, write and select are available for this connection. When the middleware is done with the connection, it calls close to release the connection.

Because AOS offers different level of security, these communication channels are encrypted using SSL [3] or TLS [RFC 2246]. A caller can itself decide what level of security is desired for a connection, by supplying a number of *cipher suite* strings, in which case the connection can only be made if both parties agree on the level of security for a connection. There must be at least one match between the cipher suites chosen for a listen endpoint, and used when calling connect. If both AOS instances also support the cipher suite, the connection can be made.

Reliable, ordered and secure connections are the only communication mechanism that AOS offers, and because a primary AOS requirement is efficiency, it would be desirable if this were a lightweight mechanism. However, SSL and TLS connection setup is rather expensive due to the fact that it requires some expensive public-key cryptography (the TLS/SSL *handshake*). If this needs to be done for every single connection (probably carrying some noncritical data) this is not good for the performance. Therefore, the AOS specification describes a lightweight multiplexing protocol (which is internal to AOS) which allows multiple middleware processes to share a TLS/SSL *base* connection. The multiplexing protocol is specified in a platform independent format, allowing different AOS implementations to communicate with each other. The AOS communication channels are also referred to as *multiplexed (MUX)* or *virtual* connections.

2.4.2 Agent Containers

As already mentioned, AOS is only the lowest layer of a (mobile) agent platform: it does not manage the actual execution of agent processes. Rather, it allows middleware systems to store and transfer the data belonging to these agents. This data can be (compiled) program code for the agent, or any other kind of data necessary for running and restarting the agent by the middleware process. For each agent, a middleware process can create an *Agent Container* (AC), which offers this storage space for the agent and its data.

Once an AC is created with the create_ac call, one can create and modify typed *segments* (essentially files) in an AC. Each part of the agent can be stored in a different segment, the type of information stored in a segment can be determined at segment creation time by the create_seg function, which takes some meta information with it in the type, subtype and description fields. Segments can also be made *persistent* to prevent any future modification of the segment, even after it is transferred.

Mobile agents usually have the property that they *migrate* over the course of their lifetime. AOS does not manage the actual running of agent processes, so the details of (re)starting agents is not something AOS does. However, as AOS offers the agent storage it also takes part in the agent transfer by offering an AC to be transferred to a different middleware process, which can be running on the same AOS or a remote AOS. This process mainly consists of converting the entire AC to a portable format, after which this data can be shipped to the other AOS to be received by another middleware process. The remote AOS rebuilds the received AC after which the application middleware can take care of restarting the agent.

Before an AC can be shipped, it must be converted to this portable format. This process is called *finalizing* and it is done by the finalize_ac call. Finalizing an AC is conceptually not that hard; it mainly consists of creating checksums (for integrity protection) for all the segments and putting all the segments in a ZIP format file. Each AC contains some special segments which are created upon finalization. These special segments are the table of contents (TOC) which contains all segment information (sizes, types, checksums etc.), a signature of the TOC for integrity protection, and a certificate containing the public key of the AOS that had signed the TOC. This information can be interpreted at the receiving AOS which can use this information to verify and rebuild the TOC. Once the AC was accepted at the remote AOS, both AOS instances now have a copy of the AC. After this, the middleware layer takes over and these decide which AOS keeps the AC and which one deletes it.

The transfer of an AC is described in the specification as the *Agent Container Transfer Protocol* (ACTP) and it is a message exchange protocol defined in the AOS specification for shipping an AC. In order for this to proceed safely, the receiver process first creates a transaction identifier (or *XID* in short) with the prepare_wait_ac function. This XID is a unique sparse identifier for a single AC transfer. The receiver can then transport this XID over a secure channel to a remote process, which can call ship_ac on its own AOS with this XID. The receiver then calls wait_ac with this same XID, and once the transfer is completed, a copy of the AC is present at both locations. Using this XID one can exactly verify that it receives an AC from a trusted source (if the XID was not stolen somewhere in between, that is).

For security, an application middleware system can decide which security suites are acceptable for the transfer of the Agent Container, similar to how this is done for the communication channels that AOS offers. Similar to the communication channels, the agent transfers can also use a single base connection simultaneously. In theory, AOS even allows regular MUX communication channels and ACTP transfers to share the same base channel. Whether it is actually allowed to mix ACTP and MUX over the same base channel is up to the actual AOS implementation to decide.

2.4.3 Role management

Because multiple application middleware systems can create resources using the same AOS process, it should not be possible to have these interfere with each other. Once a process creates a connection or an AC, it should be impossible for other application middleware processes to access these resources on AOS: the process that creates the resource is the owner, and as such is the only one that can have access to the resource. AOS uses the concept of a *role* to assign permissions and distinguish middleware processes from each other.

Roles serve as a mechanism to prevent an application middleware from calling certain AOS functions. If a process should not be allowed to call a function this limitation is associated with the role of the process. For example, for an object server that does not need to use Agent Containers it can be made absolutely sure that this process can not use ACs by disabling the AC related calls (in this case the functions 3-14 from figure 3) for its role.

AOS offers a middleware authentication mechanism using the concept of a *cookie*, and each AOS function has such a cookie as its first parameter. A cookie is a unique sparse identifier which serves as an authentication token for a process to prove its identity, such that AOS can look up its role in an internal table and see if the process is allowed to use a certain resource.

Why this is done by using a cookie is an AOS implementation decision which will be explained in detail in the next chapter on the AOS implementation. For now it will suffice to say that a cookie must be used because it is otherwise hard to distinguish different processes if a dispatcher sits between AOS and the calling process, which is almost always the case.

2.5 MSc Project Goals

The entire AOS project is aimed at providing a base for implementing a (higher level) mobile agent system. It does so by offering a basic set of functions which these middleware systems call all be built upon, even though these middleware systems can be have very different properties themselves. AOS is therefore kept really minimal, which means that preferably AOS should also be reasonably efficient. Different AOS implementations can also be made according to the AOS specification, but all different implementations should all be able to interoperate.

There is already a Java implementation of the AOS kernel, as part of the AgentScape project. This implementation is based on version 5 of the AOS specification. The main goal for this MSc project was to create a C/C++ implementation that should also conform to the version 5 API specification, and can be used in Mansion and AgentScape.

This C/C++ implementation should be reasonably portable, meaning it should at least run on most kinds of Unix/Linux platforms (currently Linux-2.6 kernels and Sun Solaris 9 & 10 have been tested). It should also be able to interoperate with the other implementations of the AOSv5 kernel, this should be tested with the Java implementation. This testing is also useful for finding inconsistencies in the API specification, as well as to discover bugs and inconsistencies in any of the two implementations.

Another motivation for programming AOS in C/C++ is performance: Java is not the fastest platform around because of its interpreted nature. C programs are compiled directly to native machine code and should probably be able to work faster, especially when doing computationally intensive work (like cryptography). An implementation of AOS in C/C++ can potentially offer a more efficient base for mobile agent systems.

The next chapters present the entire project of the AOS implementation in C/C++. Chapter 3 presents a description of the internal software architecture and design of the C/C++ implementation made for this project. This includes motivations for the design choices, advantages and disadvantages of the current approach, as well as suggestions for future optimizations. Chapter 4 contains a description of all of the testing that has been done, which focuses mainly on comparing the performance of both kernel implementations.

3 AOS Software Architecture, Design and Implementation

This chapter presents the complete architectural design of the C/C++ AOS kernel implementation. Section 3.1 presents a complete overview of the entire AOS architecture after which the implementation of the individual components of the architecture are discussed in more detail in the later sections.

3.1 AOS Implementation Architecture

We will start with a complete picture of the entire AOS architecture, which is shown in figure 4. This figure contains most of the high-level components of the AOS architecture as implemented. The individual components will all be treated later on in this section.

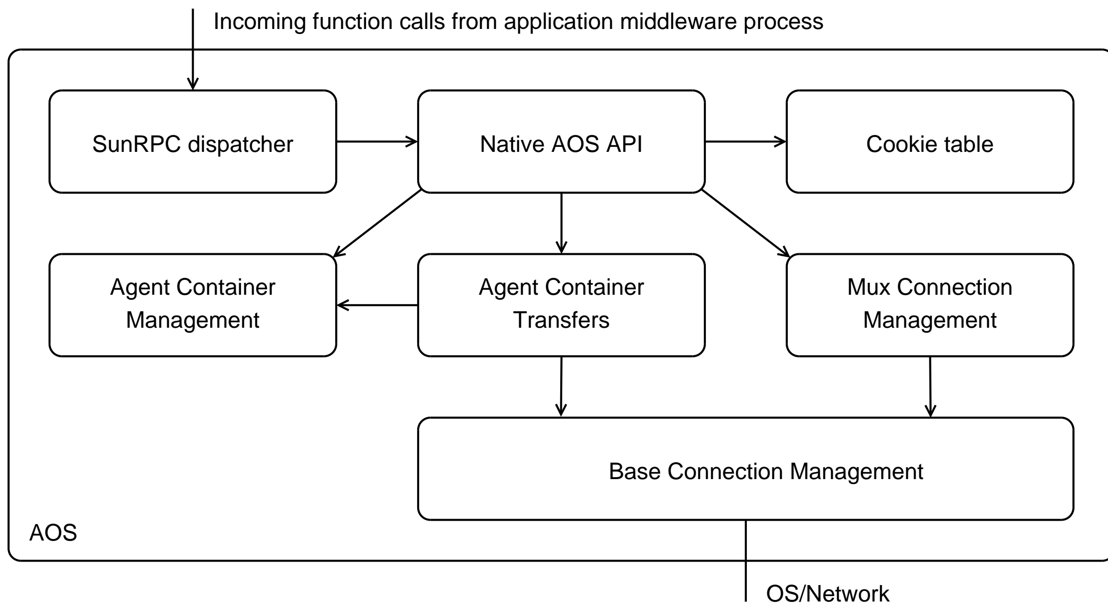

Figure 4: Architectural overview of the AOS implementation.

At the top is the only entry point: a dispatcher that receives incoming requests, and executes these function calls on the native AOS API. The native API first checks the role of the application middleware that made the function call, to see if the application middleware is allowed to perform the function call, and whether the requested resource(s) are available and owned. Once these checks are done, the function call can access the requested AOS resources (Agent Containers, AC transfers or connections). The native API calls the requested method on the AC management module, the AC transfer module or the communication module. A more detailed figure of the architecture is presented in figure 5, which is included for reference. Details of the individual modules are discussed later in this chapter.

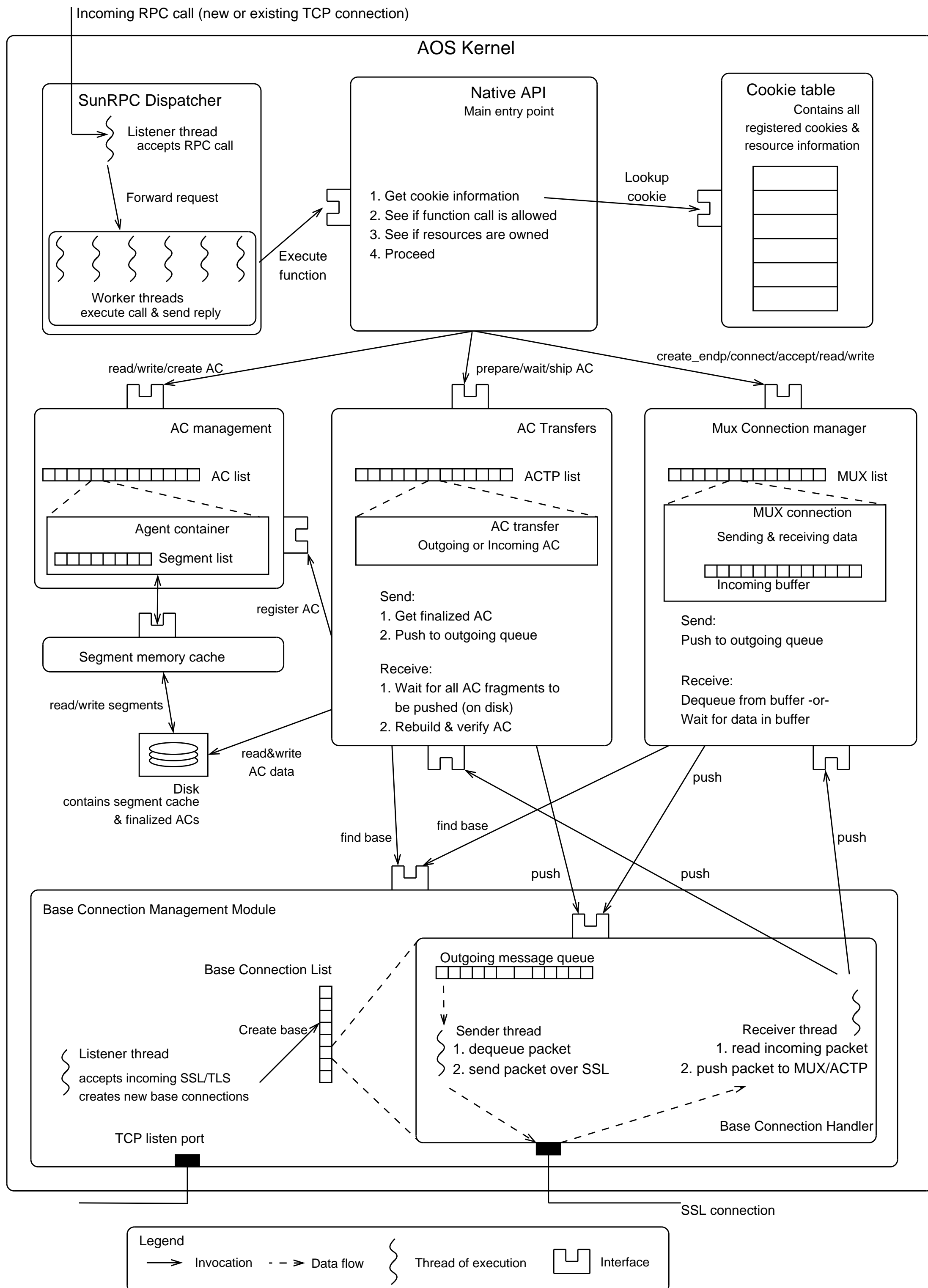

Figure 5: Complete AOS implementation architecture

Figure 5 shows more detail than figure 4, most notably the interaction between the internal modules, as well as more internal detail of the individual modules themselves. This complete figure is included for completeness, and can be used as a reference of the complete architecture, but the individual modules are all described in this section.

Figure 5 also contains the different threads that individually run inside AOS. For the dispatcher there is one single handler thread that reads incoming RPC requests. Once such a request has arrived, the function request is translated to a native function call and executed by a dedicated worker thread. This worker thread comes from a pool, and all workers just wait for jobs to be put into the job queue which they can execute. After the worker finishes a request, it sends the result back over the IPC channel from which the request came in. The worker thread is then ready to handle a new job.

Each base connection has two handler threads as well, one for sending outgoing messages from the MUX and ACTP connections that use them, and one receiver thread that fetches messages from the SSL connections, which it then forwards to the appropriate MUX or ACTP connection. The base connection has a simple *push* interface which allows MUX and ACTP to enqueue messages which will eventually be sent out by the sender thread. Similarly, the MUX and ACTP modules have a push interface, which allows the base connection to forward the messages it receives over the base connection to the correct module. There is also a single AOS listener thread which accepts incoming SSL/TCP connections (currently, AOS has a single listen endpoint), and a special garbage collector thread (not shown) which takes care of cleaning up unused resources.

This chapter describes the architecture of the dispatcher (3.2), the organization of the cookie and role information (3.3), the layout of the agent container management module (3.4), agent transfers and multiplexed connections (3.5), cleanup of unused AOS resources (3.6) and crash persistence of cookies and resources (3.7).

3.2 Dispatcher

The only dispatcher present in the C/C++ implementation is a SunRPC dispatcher running over TCP. SunRPC was chosen because it uses XDR which is a relatively fast data conversion (much faster than ASCII-based XML-RPC at least) and it is very basic so that it can be used by agent application middleware systems in different programming languages, which is one of the main AOS requirements (see section 2.1).

Some functions of the AOS API can block, but because multiple middleware processes can use the AOS process at the same time, it is important that the dispatcher itself does not completely block until some other blocking function completes. Should the dispatcher block, then all other processes can not use AOS until the blocking function completes (which can take forever).

A number of approaches are possible to allow concurrent access of the dispatcher. One is to have a dispatcher with threading support in which one listener thread takes care of reading the RPC request, and the actual function execution is performed by some other thread. Another approach would be to use some kind of asynchronous request processing (e.g. the calling process can receive its data later without blocking the dispatcher) or to use a callback interface so that AOS can send back the results of the function to the middleware process as soon as the function completes.

Since AOS already requires threading support for handling incoming and outgoing data on base connections, the current implementation of the SunRPC dispatcher also uses a threaded design. The design of the current implementation of the dispatcher is shown in figure 6. This figure shows that there is one listener thread which reads the incoming RPC requests, and a pool of worker threads that execute the function calls on the native API. Once the listener receives a RPC request, it puts this request information in a queue. One of the worker threads will eventually dequeue and process the request, after which the worker sends the result of the function back over the TCP connection over which the request arrived. The SunRPC library keeps these TCP connections open for later use.

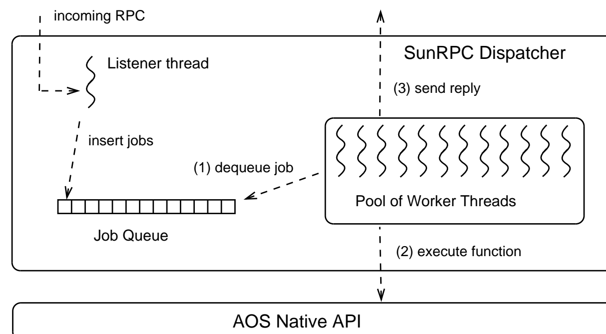

Figure 6: SunRPC dispatcher implementation architecture

A disadvantage of using a pool of worker threads is that at some point all of them can be busy processing requests. Should all of the worker threads be busy, the dispatcher can (temporarily) create a new worker thread if the job queue grows too large.

A lot of care has to be taken to ensure that different worker threads do not interfere with each other. On Linux systems, the SunRPC functionality is part of the standard C library (libc). Unfortunately this library is not in the least thread-safe when it comes to SunRPC. Handling these limitations required locking around the TCP socket file descriptor. This required inserting modified code from libc into the SunRPC dispatcher code, to set the lock for a file descriptor early enough.

However, a workaround for SunRPC modification is not needed for all platforms. The RPC library on the Sun Solaris operating system has an automatic multithreaded mode, and the library that contains the SunRPC functionality on Solaris is already thread-safe. Nice as this might seem, this is not a solution that works on all platforms. To support other platforms, the threads have to be manually managed with the pthreads library. The current implementation uses pthreads and also compiles and runs correctly under Solaris.

More details concerning the implementation of the SunRPC dispatcher (from a programmer's point of view) can be found in appendix B.

3.3 Cookie and Role Management

The concept of a cookie was introduced in section 2.4.2 on role management. A cookie is an unique sparse identifier for a process (or a group of processes) associated with a set of AOS

resources and permissions (a role). A cookie is used to identify middleware processes that make function calls. Agents or other processes using an agent application middleware do not interact directly with AOS but with the middleware. The application middleware process calls AOS functions: AOS only receives calls from the middleware processes.

Associated with each cookie is a list of functions (represented by a bitmap) that the owner of this cookie is allowed to call. This is called a *role*. Should a process not be allowed to use Agent Containers for whatever reason, the calls for creating and modifying ACs can be disabled for this role.

Conceptually, a cookie represents a single entity (or group) owned by a principal. AOS uses this cookie as an authentication token for the application middleware process. AOS does not manage cookie distribution: an application middleware process can create a new cookie and pass it to another application middleware process (preferably in a secure way), for example when a new process is started. This is especially useful if the application middleware consists of multiple processes which all need to access AOS. In this case, the initialization process of the application middleware can create a cookie (with different roles) for each component that must have access to AOS.

Allowing application middleware processes to dynamically create a cookie whenever they desire makes cookie usage flexible: different components of the application middleware can all use a different cookie, or they can all use the same cookie. Using different cookies for all application middleware processes offers a mechanism to ensure that no processes can interfere with other application middleware processes that hold different cookies. Using a different role for each application middleware process also makes it possible to prevent access to specific AOS functions.

Cookies can be created dynamically using the AOS create_role primitive. Using create_role, an application middleware process also sets the role of the child cookie, by supplying a role bitmap (which contains information as the which functions the child cookie is allowed to call). The process that creates a cookie can pass the cookie it created on to another process so that the new process can now access AOS using this cookie.

Consider, for example, an application middleware system that consists of an object server and an agent server, which both run as separate processes. The initialization process of the application middleware system can then create a cookie for both processes: a cookie that has the calls for agent containers disabled for the object server, and a cookie that is allowed to call communication and AC related functions for the agent server. The initialization process can also choose to disable the create_role function for the cookies it creates, so that the processes receiving the cookies can not create new cookies and roles themselves.

The role and cookie assignment is entirely up to the application middleware processes, and the creator of a cookie may keep a copy of the cookie for reference if it wants. The creator of a cookie may also delete this cookie at some point in time, usually when the process that possessed the cookie exits. An application middleware process may hold multiple cookies or can choose to have cookies shared between different application middleware processes. Note that different cookies may have the same role.

The cookie of an application middleware process that creates a new cookie is called the *parent* cookie of the newly created cookie. The parent-child relations of cookies are registered in AOS. A cookie's role cannot be changed after creation time. The role of a child

process cannot be less restrictive than the role of its parent cookie. When exactly a cookie is to be created is up to the application middleware processes. Upon AOS startup there is only one cookie registered in AOS, which is the cookie for the *init* role. AOS exports this cookie by some external mechanism (e.g. a file) so that another middleware process can read it. Usually there is a single initialization process that can read this cookie and start all other application middleware processes that should be able to access AOS. Init can create a new cookie for each middleware process with the create_role call, which creates a child cookie and role that can access AOS.

An owner of a valid cookie can create ACs, listen endpoints, connections and XIDs (if its role allows for invoking the corresponding call). Such resources created in AOS are associated with a single cookie: the cookie that was used in the call to create the resource. Cookies are therefore the mechanism to prove resource ownership to AOS. Access to and modification of these AOS resources will only be possible if the same cookie as the one that was used to create the resources can be presented. Cookies therefore also provide a mechanism to allow different middleware processes to work in their own private data space in AOS, without interference from other middleware processes.

A cookie is required for AOS to do role management, because there is a dispatcher layer in between AOS and its client processes. AOS cannot use persistent TCP connection information to implement an authentication mechanism, as it may not always use a connection oriented transport, which means that it is not enough to authenticate an application middleware process only once, because each function call can come from an arbitrary process. Instead, the application middleware process must be authenticated for every function call. This is why the AOS specification uses a cookie for each function call. This also means that authentication is independent of the dispatcher implementation. This means that the dispatcher can be as simple as possible as it does not need to know which process or cookie actually made the function call on AOS. The dispatcher simply calls the native API with this cookie as a first argument of the method, and the native API then takes care of checking the cookie. This makes it easy to add other dispatchers types, because their functionality is quite minimal.

Because the application middleware processes are in charge of distributing the cookies, this also means that processes that are not known to the middleware initialization processes can not access AOS.

3.3.1 Cookie Table

Once a request arrived through the dispatcher and it enters the native API main entry point, the cookie is checked (regardless of which function is executed). The cookie table is the place where all this information is stored, entries can be obtained by searching for a matching cookie entry in this table. Because cookies are unique identifiers, there should be only one entry at maximum. An illustration of a table is shown in figure 7.

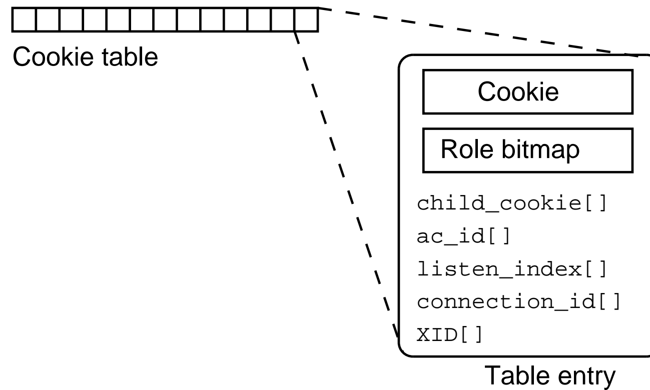

Figure 7: Cookie table entry

The cookie table is an unsorted list of cookie table entries. If an entry for the cookie provided in the function call does not exist, then the cookie is not registered and AOS immediately returns an error. If the cookie does exist, then a check is made whether the cookie actually allows the function call by checking the role bitmap. If the bit for the corresponding function is not set, then AOS will not execute this function and immediately returns an error.

Finally, resource ownership is checked. Which resource has to be checked depends on the function call. The identifiers for each resource (integers for ACs and connections, references to child cookies and XIDs) are directly stored in this cookie table entry. Each AOS function call has only 1 AOS resource as its argument, so this can be checked quickly. If the resource identifier is not registered with this cookie, AOS returns an error indicating that the resource does not exist.

Once all the checks are made, the call can proceed. This means that the underlying layers of AOS do not know anything about cookies: they assume that all the cookie checking passed at the main entry point. If a call passes this entry point, everything is assumed to be fine.

3.4 Agent Container Management Module

Agent Containers are the main storage space offered to middleware processes. The concept of an AC was introduced in section 2.2. It basically offers mechanisms to assign a portable file system to a running agent. An AC can be shipped to a remote AOS where it is used to restart an agent process. The AOS specification does not really prescribe how segments are to be stored in AOS, but it does mention the format of a finalized AC.

An agent container can be created with the AOS call create_ac, which returns an AC ID. This ID is essentially nothing more than an identifier for a list of segments, so the segments created in different ACs will not interfere. A client process can then use the functions 9-12 (figure 3) to manipulate segments for this AC. These calls are similar to those used to manipulate regular files. Segments contain raw data (uninterpreted by AOS) which is stored internal to AOS. Because AOS must be able to determine exactly which segment is requested, all segment related calls contain both the identifier for the AC and the identifier for the segment.

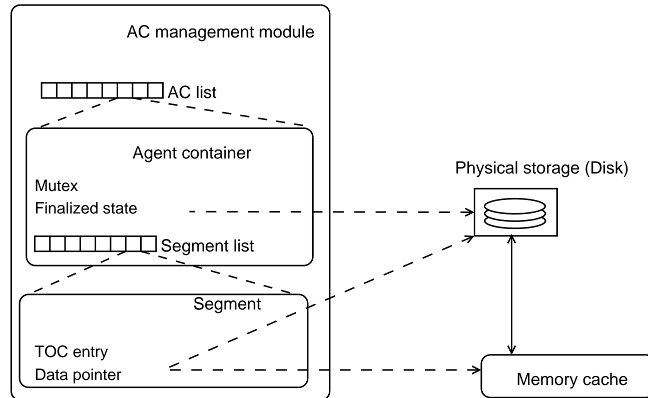

Figure 8: AC organization

Figure 8 shows the data structures the Agent Management module uses internally. As can be seen from this figure, the AC management module contains a list of all agent containers. Each AC has at least a list of segments, a mutex (for concurrent access) and a reference to the last finalized state on disk (if any). Unless noted otherwise, each AOS data structure has a dedicated mutex to prevent race conditions. Segments, however, do not have a dedicated mutex, but they are protected by an AC-wide mutex. A per-segment mutex can be added with only a small modification to the AOS code, if needed (see appendix C). The data for each segment can be stored on disk, or in a segment memory cache, depending on the memory use of AOS. More details about segment caching are described in section 3.4.2. Segments also have a TOC entry which can be inspected by the AOS `read_toc` function. They also have a pointer to the data for the segment (a location in memory or on disk).

3.4.1 AC Finalizing

At some point, in particular when an agent wants to migrate, all AC data must be finalized. The idea behind this is that once an agent container is finalized, there is a persistent (and integrity protected) snapshot of the current state of the agent on disk. This not only allows for migration (using the `ship_ac` function) of the AC, but also for AOS crash persistence (by storing the AC on disk). The AOS specification requires a checksum for each segment, and a list of all TOC entries (this list is stored in the AC with segment ID 0). Also, a signature has to be created for this TOC, to protect against tampering with the AC/TOC contents. This signature is stored in segment 1. The certificate of the AOS that signs the AC is then stored in segment 2. If segments 0-2 already exist, their contents are replaced with this new information. AppMW and/or AOS may keep an extra copy of these first 3 segments before they are replaced as part of an audit log. Once this is all done, all segments are stored in a ZIP file, after which the finalization process is done. Finalizing in the C++ implementation of AOS consists of the following steps (in this order):

- Update all segment checksums (for modified segments only)
- Generate table of contents (TOC), store this as segment 0 (on disk)
- Sign TOC with the AOS private key and store the signature in segment 1 (on disk)
- Put the AOS self-signed certificate in segment 2 (on disk)
- If present, backup (rename) previous finalized state as oldac.zip for atomicity
- Store all segment files in the file ac.zip on disk
- Synchronize (fsync system call) the file ac.zip to disk.
- When finished
  - If successful, remove oldac.zip
  - On failure, rename the oldac.zip file to ac.zip

This is perfectly straightforward and according to the finalization process described in the AOS specification. The TOC and signature segments are created, and all segments are added to the zip archive. Should anything fail when creating the ZIP file, then the most recent finalized state can be restored by renaming oldac.zip back again to ac.zip which was done to ensure that finalize is indeed an atomic operation: it either succeeds or it completely fails. Upon failure, the old finalized state is automatically recovered by AOS.

Each AOS instance uses a dedicated working directory, of which the location is determined at AOS startup time. A finalized AC is always stored at a well-known location on disk relative to this working directory as agentcontainers/<ac-ID>/ac.zip. Segments stored in the disk cache can also be found in this directory as agentcontainers/<ac-ID>/<segment-ID>.

An early version of the C/C++ kernel implementation relied on an external zip program to create the ZIP archive. This approach required all segments to be stored on disk before the archive could be created. The current implementation relies on the ZipArchive library (see appendix A) which supports in-memory compression and storage to reduce the I/O overhead.

ZIP offers not only data storage, but also data compression. One can choose a desired level of compression when storing files in an archive, which can be 0 (no compression) or a value between 1 (fastest, least compression) and 9 (slowest, best compression). Usually, fast compression results in a file that is already a lot smaller than the original file. The differences in size between the other compressions levels are fairly insignificant, although the compression time does increase. The default level of compression used in AOS is the fastest level (1), because it takes the least extra time compared to level 0, and results in a significant reduction in data size. Another value can be set in AOS at compile time if desired.

There are some more optimizations that could have been made, but which did not make it due to a lack of time. For example: when an AC is finalized multiple times (probably over multiple AOSs) only the modified segments really need to be compressed again. The old segments are still present in the original ZIP file, which could save some time. For example: if an agent takes 10 hops over 10 different AOS instances, and it adds 5 megabytes to the AC on each hop, only these 5 megabytes need to be compressed on this new hop. This would make the finalizing more efficient, but it may be a little harder to ensure atomicity of the entire operation. The entire finalize process has to be verified for correctness again. In its current implementation, AOS compresses all segments to a new file upon finalization.

Another possible optimization might be working directly from the ZIP file, or storing persistent (read-only) segments directly in the ZIP file. Again, this optimization may make it a little harder to obtain atomicity (because then we need to distinguish between the file containing the finalized state, and an archive containing a copy of this used at runtime).

Although some optimizations might be possible, finalization is still an expensive operation. There are many cryptographic operations involved, and when wanting to be absolutely sure that the agent container is physically stored on disk (not just in the disk cache) then the archive also has to be synchronized to the disk device (by using fsync). All operations increase the time needed for finalizing. There is not really a lot that can be done to speed this up: if it is to be done in a safe and secure way there is a price to be paid.

3.4.2 Segment Caching

AOS maintains an internal segment cache. Earlier versions of the C/C++ implementation relied completely on internal memory for storing segments. Although using internal memory for segments is usually faster than maintaining a segment cache on disk, this is not a complete approach: internal memory is always limited. So if AOS supports many application middleware processes, it may not be able to cache all segments in memory, even when taking memory swapping into account (swap space is usually also limited).

The Java implementation uses a different approach: it keeps all segments entirely on disk. This means that the Java implementation has a lot more storage space available for segments. In terms of performance this will probably not be optimal.

The current version of the C/C++ implementation uses a combined disk and memory caching scheme for segments. A segment can either be stored in memory or on disk. The basic structure of a segment descriptor in the current implementation in a C++-style description is:

```
class segment {
   TOC_ENTRY segment_id;
   bool in_memory;
   bool modified;
   List *buffer;
}
```

In the current implementation, most of the internal data structures are modeled as C++ classes. These classes also contain extra measures like mutex protection and more but these are not shown in this example.

In the earlier AOS versions that did not have a disk cache, the entire segment was stored in the memory buffer (a linked list so that the buffer could dynamically grow). This was changed so that AOS could also use the disk for segments. However, some extra administration is needed to monitor the total size segments occupy in the memory cache. The segments that are in internal memory still have their own buffer (there is no such thing as a central memory pool), but the implementation maintains a central counter that keeps track of the total size of all segments currently present in internal memory.

Because there is only one central counter that keeps track of the amount of cached segments, each operation that requires more memory for segments must explicitly increment this counter, so as to register the amount of memory the segment occupies.

There is an internal limit on the total size of the memory cache, which can be set in AOS at compile time. If a function appends data to a segment, the extra bytes have to be added to the memory cache counter. If the counter is still below the limit, it is possible to have more bytes for the segment in memory. If the counter indicates that the memory cache is full, then the segment cannot be kept in memory and it must be moved to disk. The interface for the (thread-safe) segment counter is described in the following table.

Function	Action
bool **try_register_bytes** (int)	If the extra bytes can be added to the memory cache then these bytes are added and the function returns true. Otherwise it returns false.
void **deregister_bytes** (int)	Removes the registration of bytes from the memory cache. After the segment is moved to disk, these bytes can be used by other segments

Figure 9: Global segment counter interface

The functions that need to allocate extra space for their segments must explicitly use the functions for updating the segment cache counter. If the try_register_bytes function returns true, this means that the extra bytes can be kept in memory. If there is no space for the extra bytes then this function returns false, and the calling function must move the segment to the disk cache before it is allowed to append extra data to the segment. So effectively, the counter is the only central mechanism of the segment cache, with the memory cache itself being distributed over the individual segments. A description of how this is done in the AOS implementation is shown below in pseudo code for the functions that interact with the segment cache counter.

```
try_move_segment_to_memory () {
  /* a segment can only be moved to memory if extra space is left */

 if (segment_on_disk() && segment_can_be_cached() && try_register_bytes (segment_size))
   move_to_memory ()
}

bool segment_can_be_cached () {
  if (segment not too large) return true;
  else return false;
}

seg_write () {
  if (segment on disk)
    try_move_segment_to_memory()

  if (segment in memory) {
    if (write_increases_size && (! try_register_bytes(extra_size))) {
      move_segment_to_disk ()
      deregister_bytes (segment_size)
    }
  }
  write_segment_data () // either writes to disk or to memory

  if (! segment_can_be_cached())
    move_to_disk ()
}

seg_read () {
  if (segment_on_disk)
    try_move_segment_to_memory()

  read_segment_data () // either reads from disk or from memory
}
```

Both seg_read and seg_write try to move the segment to the memory cache before attempting to access it by calling the try_move_segment_to_memory function. If a segment is on disk, this function tries to register the extra bytes needed for the segment with the cache counter. If this is successful, the segment is moved to the internal memory. Otherwise, interaction with the segment has to be done directly from disk.

Should a write operation on a segment increase the total size of the segment, then these extra bytes also have to be registered. If the extra bytes fit in memory, then the operation can proceed. If the bytes cannot be added to the segment cache counter, then the segment must be moved to disk before the write operation can be done.

This caching structure implies that once the memory cache is full, new segments must always be manipulated on disk until enough memory space becomes available again for the segments. So effectively this caching uses a first-come-first-serve policy. This is not optimal because this scheme does not take segment usage frequencies into account. A least-recently-

used (LRU) scheme might be better, but this means that the segment storage has to be done differently (centralized). The current scheme was chosen for its simplicity: it was a relatively easy way to modify the original (internal memory only) caching scheme. LRU is probably a more fair way of sharing the segment memory cache and is a recommended future optimization.

However there are some optimizations that were added in order to prevent segments from occupying too much of the internal memory. The segment_can_be_cached function checks for this. Currently, whenever a segment is considered to be too large it can never be stored in the memory cache. The limits of the total memory cache size, and the maximum segments size that can be cached can be defined in AOS at compile time. Persistent segments are also never cached but directly read from disk. Once a segment is made persistent, its checksum is created and its contents are moved to the disk cache.

Currently the limit for the total memory cache size is 256 MB. Segments that become too large (currently this limit is set at 1 MB) are directly accessed from disk, but these values can be changed at compile time in the main AOS configuration file.

Another optimization that can be enabled at AOS compile time is completely moving an AC out of the internal memory once it is finalized. However, this is somewhat expensive because it introduces extra I/O overhead. To solve this, a background thread is started after finalization completes. This thread moves all segments to the disk cache, so these disk operations are not added to the total times a finalize operation takes. This strategy is based on the assumption that these segments will probably not be used soon after finalizing, so it may be best to move these segments to disk to make room for other segments, though finalizing an AC does not strictly mean that all segments will not be read anytime soon (e.g. when ship_ac fails the AC contents may be needed again for restarting an agents).

3.5 Connection Management

Another large portion of functionality that AOS offers concerns connections. Internally, AOS has two different kinds of connections, which are *base connections* (the actual SSL/TCP connections) and *virtual connections.* Virtual connections are multiplexed communication channels running on top of a base channel. Both AC transfers (ACTP) and regular communication (MUX) are multiplexed over base channels. This section describes the design and architecture of these connections, including the implementation of flow control, and the administration of virtual connections over base channels.

The entire multiplexing protocol is specified in the AOS specification in XDR format. This includes the semantics for connection setup, connection termination and payload. Both ACTP and MUX messages are encapsulated in what the specification calls *proto* messages. In XDR this is defined as in the following structure:

```
enum PROTO_ID { MUXP, ACTP };   // all the supported protocols

struct PROTO_MSG {
   unsigned int proto_version;
   unsigned int body_length;

   union body switch (PROTO_ID p) {
      case MUXP: MUXP_MSG muxp_msg;
      case ACTP: ACTP_MSG muxp_msg;
   }
}
```

The union contains either a MUXP or an ACTP message. Because base channels encapsulate all protocol-specific messages (including payload) in proto messages, this means that it is in theory easy to mix ACTP and MUX over a single base channel, though it depends on the implementation itself whether this is actually allowed. The Java implementation distinctly separates ACTP base channels and MUX base channels (it returns an error if an ACTP message is sent over a base channel that is used for MUX and vice versa), the C/C++ implementation can handle both types running over the same base channel. The connection over which these PROTO_MSG messages are sent are standard TLS/SSL connections over TCP, which can be used between different kind of platforms.

The definition of the MUXP (multiplexed connection protocol) and ACTP (Agent Container Transfer Protocol) messages are also defined in the XDR specification. In short, a MUXP message can either be an *open*, *open-ack*, *open-nack*, *payload* or *close* message. An ACTP message can be a payload (*ship*) message, or a *reply* message.

3.5.1 Base Connection Architecture

Part of the base connection management structure is shown in figure 4. Basically, there is one listener thread that waits for incoming SSL connections. If such a connection comes in, a base connection entry is registered. This base connection can be used for incoming as well as outgoing virtual connections. Base connection entries are created whenever a virtual connection must be made to an AOS for which no (suitable) base connection already exists. How this exactly works is explained in the next figure:

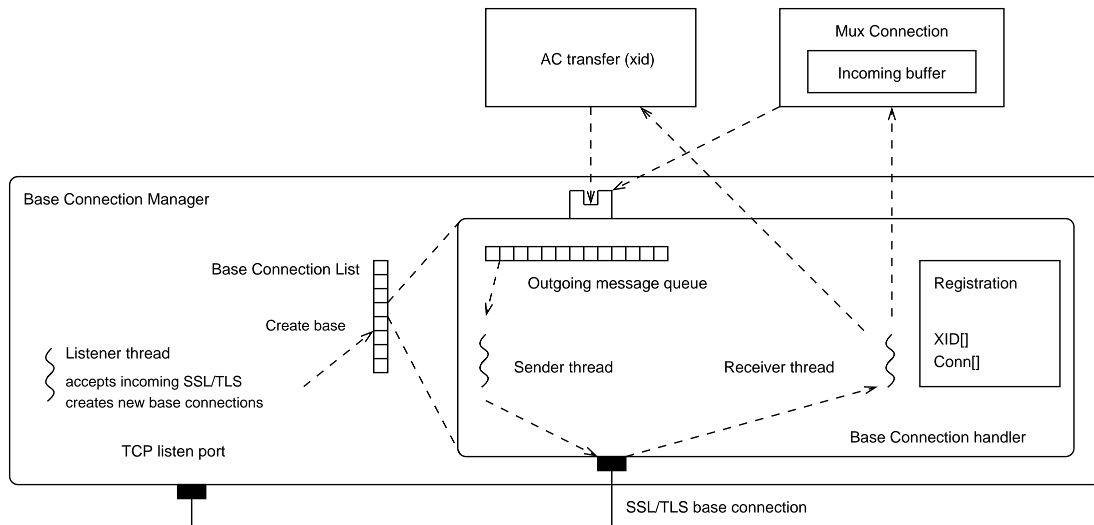

Figure 10: Base channel architecture

Figure 10 depicts a single base connection handler with one ACTP and one MUX connection running over it. As this figure shows there are two active threads for each base base connection handler. The interface of the base connection handler for sending data is simple: the only allowed function is to push a proto message to the outgoing queue. Receiving data from a base connection is done passively, by waiting for the receiver thread to push the data back to the corresponding MUX or ACTP transfer. There is an incoming buffer for each MUX connection containing the data that was received from the base connection.

All base connection handlers are part of the base management module, but each handler itself is really a self-contained module as well, because each base connection handler works on its own (because of the sender and receiver thread) and has a dedicated interface. The threads are started up at the time of base connection handler creation (create base in figure 10). Each base connection handler is a C++ object which keeps track of the sender and receiver threads and connection administration itself.

Each base connection handler also contains a list of registered virtual connections. Should the underlying SSL connection break for some reason, the base connection notifies all registered connections of the failure. This means that if there is still data pending in the outgoing queue, this data cannot be sent, even though the sending party already assumed that everything went well. This is a disadvantage of using an outgoing message queue, based on the underlying SSL/TLS transport semantics for sending outgoing data.

3.5.2 Virtual Connection Setup

Each virtual connection requires a base connection on which to run. Essentially, base connections are just SSL connections over which proto messages (containing MUXP and ACTP as payload) are sent and received. The problem is that each application middleware process can only define the acceptable encryption for the connection on its own side of the connection. Based on this information, AOS can find a matching base connection over which the MUX connection can be made. The problem is that the application middleware process

that accepts the connection can have different requirements for security (specified at endpoint creation time). Even if there is an intersection between the cipher suites that are acceptable for both connections, the base connection used to set up the MUX connection may not be acceptable for both initiators of the connection.

An example that includes the problem of a virtual connections setup over an unsuitable base connection is illustrated by the following figure.

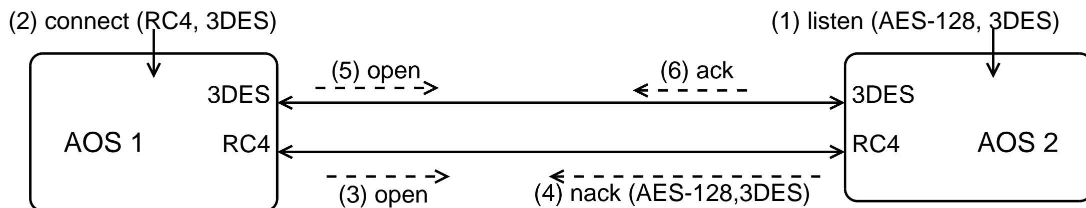

Figure 11: Example of virtual connection creation

Figure 11 shows two running AOS instances connected by two different SSL connections, each using a different cipher: one is encrypted with RC4, the other uses 3DES. In this example, a client running on AOS 1 attempts to connect to a process on AOS 2.

Before an application middleware process can setup a virtual connection, there must be an endpoint created on AOS 2. This is done with the create_listen_endpoint (step 1). This function includes, among others, a series of cipher suites which are known to be acceptable for this connection. In this case the application middleware process is willing to accept connections using 3DES and AES-128, which means that a virtual connection can only be accepted if the virtual connection was made over a base channel running either one of these acceptable cipher suites.

Step 2 is for the client on AOS 1 to create a connection to this endpoint using the AOS connect call. This requires a set of security suites. For this example, the process allows RC4 and 3DES to be used.

AOS 1 is already connected to AOS 2 by two base channels, which both have acceptable cipher suites for the connect call. So AOS 1 can try to setup a connection by sending a MUXP open packet to AOS 2 over the RC4 encrypted connection (step 3).

However, the application middleware process that created the listen endpoint (that runs on AOS 2) does *not* allow a connection to this endpoint to use the RC4 cipher. So even though the connection was valid for the connecting process, a connection cannot be made because the accepting application middleware process does not allow this cipher for its connections. However, AOS 1 does not know beforehand that it could not have used the RC4 channel, because the allowed set of security suites for the application middleware listen endpoint is only known at the target AOS.

This problem is what the AOS specification calls a *base connection security suite mismatch*, caused by the fact that the AOS listen endpoint for the virtual connection is hidden behind the SSL/TCP connection endpoint. This means that a target AppMW endpoint may not find the base channel encryption suitable, even if the connecting AppMW did. Should this

happen, AOS 2 sends back a negative acknowledgment (step 4) containing the reason why the connection setup failed, and if this was caused by a cipher suite intersection, the nack packet contains the list of cipher suites which are acceptable for connecting.

AOS 1 can see if there is a match between the security suites in the nack packet and the security suites passed to the connect. If there is no intersection then the connection can never be made with these cipher suites and an error is returned to the caller. In this example there is an intersection (3DES is supported by both) so AOS 1 can retry to create the virtual connection over a suitable channel (step 5). In this example, a suitable 3DES encrypted connection already exists, over which the connect call is retried. Now both the connecting middleware processes and the listening process agree on the cipher suite to use, and the connection setup is completed by AOS 2 sending back an acknowledgment packet (step 6).

There are a number of other reasons why a virtual connection setup can fail, for example if one of the AOS instances is already running ACTP over a base channel and it does not allow MUX and ACTP to be mixed. It can also be that one AOS has a maximum number of virtual connections that can run over a single base connection, but the other AOS does not have such a limit. Error codes for such cases are defined in the multiplexing protocol. When AOS receives such an error, it can decide whether it can try to use or set up a different base connection, give up or retry later.

3.5.3 Base Connection Selection

If AOS wants to set up a virtual connection (for MUX and/or ACTP) it is the task of the *base connection management module* to see if suitable base connections over which the virtual connection can run already exist. The base connection management module retrieves a list of all existing connections for the requested AOS (indicated by the AOS *endpoint*, a combination of an IP address, TCP port and AOS ID). The interface for this module is shown in the figure 12.

Function	Action
int[ ] **find_bases** (endpoint ep, char[ ][ ] cipher_suites);	Returns a list of all the available base connections that are connected to AOS identified with endpoint ep using any of the cipher suites in cipher_suites.
int **base_connect** (endpoint ep, char[ ][ ] cipher_suites);	Requests the module to actively create a new base connection to endpoint ep, using one of the suites from cipher_suites.

Figure 12: base connection management interface

The organization of the base connection manager is shown in figure 13. The base connection management module contains a list of all registered base connections, including the AOS kernel IP address and port to which they are connected, and the active cipher used for this base connection. The base manager uses this information to make a selection of suitable base connections when the find_bases function is called.

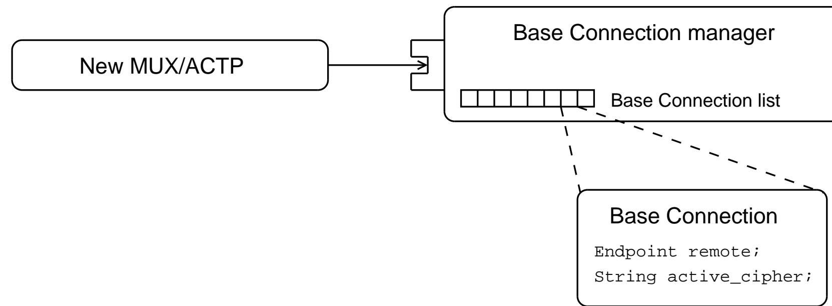

Figure 13: Base channel selection

When a process wants to set up a virtual connection, it can do so by using find_bases to retrieve a list of suitable base connections. If no suitable base connection is found (or connection setup fails over all existing base connections) the base manager is requested to actively create a new base connection with the appropriate properties. The algorithm used for finding base connections for MUX connection setup is relatively simple, as shown below.

```
int[ ] find_bases (endpoint ep, char[ ][ ] allowed_suites) {
    int[ ] result
    for (all base connections) {
        if (base.remote == ep && (allowed_suites contains base.active_cipher)
            add_baseid_to_result ()
    }
    return result
}

mux_connect (endpoint ep, char[ ][ ] suites) {
    int[ ] baselist = find_bases (ep, suites) // try existing connections first

    for (int i = 0; i < baselistsize; i++) {
        if (try_muxconnect (baselist[ i ]) == OK)
            return OK;
        else if (intersection_error)
            return mux_connect (ep, intersection-suites); // recursive call
    }
    // a suitable base connection does not exist, try to create a new one
    int baseid = base_connect (ep, suites)

    if (try_muxconnect (baselist[ baseid ]) == OK) return OK;
    else if (intersection_error) return mux_connect (ep, intersection-suites); // recursive call
}

bool try_muxconnect (int baseid) {
  /* this function tries to setup a virtual connection over the base connection “baseid” by
     performing the MUX open/ack message exchange */
}
```

The details of ACTP transfer setup are a little different, but base connection selection is similar. First, a list of available base connections is obtained, which are all tried one at a time. If one of these results in a correct connection setup then the connection is made. If an error occurs, the virtual connection setup cannot be used over this base connection. Should the received error indicate that there is a security suite intersection, the connection setup is retried for each of the base connections that use security suites in the intersection.

If MUX connection setup fails for all of the the existing base connections, then AOS requests that the base manager makes a new connection. If this succeeds then this function returns successfully. If connecting fails there can still be a security suite mismatch. If this is the case, the connection is retried with the a set of cipher suites, otherwise the function fails.

3.5.4 Virtual Connection Administration

Whenever a virtual connection is successfully made, this connection is registered with the base connection using a reference counter and a list of virtual connections identifiers that are running over the base channel. This reference counting is done transparently: the base connection understands all messages for the virtual connection that are sent or received. To sum up, the base connection takes the actions described in the following figure, according to the messages it sees passing (either incoming or outgoing).

Protocol	***Message***	***Action when message comes by***
MUX	OPEN	No registering (wait for either ACK or NACK)
	OPEN-ACK	Register virtual connection (now payload can be sent)
	OPEN-NACK	Do not register (connection failed)
	PAYLOAD	Only acceptable if connection is registered
	CLOSE	Remove registration of virtual connection
ACTP	SHIP	Register XID present in ACTP message
	SHIP-REPLY	Remove registration of XID

Figure 14: Message types (from specification) and associated actions

The description of what these MUX connection setup messages precisely mean can be obtained from the AOS specification, but it is essentially a two-way open/ack construction for connection setup, and a symmetrical close for termination of the connection.

ACTP transfers are a little different, as an entire AC transfer consists of a batch of ACTP ship messages, and a single reply. If a ship message with a XID comes by, the reference counter is only updated if this is the first message that was sent using this XID. Whenever an ACTP ship reply message is seen this always indicates the end of the ACTP transfer.

ACTP connection setup is different from MUX connection setup, the next figure describes the semantics for initiating an ACTP transfer.

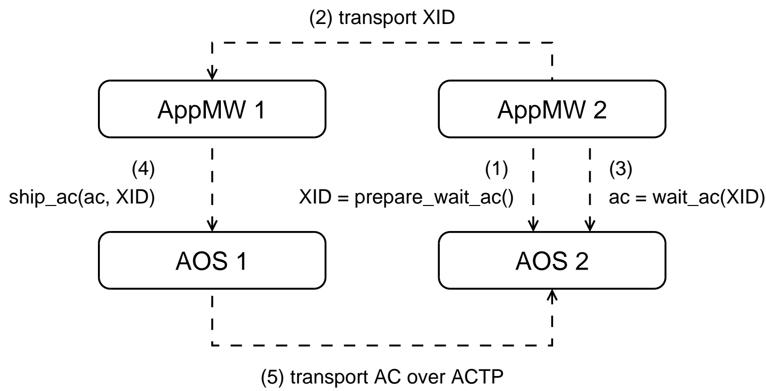

Figure 15: ACTP transfer semantics

Figure 15 shows which steps have to be taken in order to successfully complete an entire ACTP transfer. First, the process that wants to receive an AC creates a transaction identifier (XID) by calling the AOS prepare_wait_ac function. This XID is a unique (sparse) identifier which can be used to identify a single ACTP transfer. AppMW 2 uses an earlier setup (secure) channel (generally created by the AppMW 1 that wants to ship an AC) to transport this XID to the process which will send the AC (step 2).

After transporting the XID, AppMW 2 waits for ACTP to complete for this XID by calling wait_ac (step 3). The sending process on AppMW 1 then sends the AC out with this exact same XID (step 4) after which the ACTP runs until completion, and wait_ac returns after the AC has been received and verified.

3.5.5 Local Communication

If an application middleware process wants to communicate with an endpoint located at the same AOS, then it is a little expensive to use SSL connections for this, because no network communication has to be done between two AOS instances. Setting up a SSL base connection over the loopback IP is possible, but it is a little expensive as this requires a new SSL connection and handshake. Using SSL for this does not increase security because both middleware processes already trust the local AOS. This local communication is considered to be at least as secure as the strongest SSL encryption.

So instead, for local AOS communication, a pseudo base connection is used, which does nothing more than push the enqueued proto messages back to the local AOS. Because no encryption is used, the cipher suite arguments passed to the communication set up calls can be safely ignored. The layout of this single local connection handler is shown below.

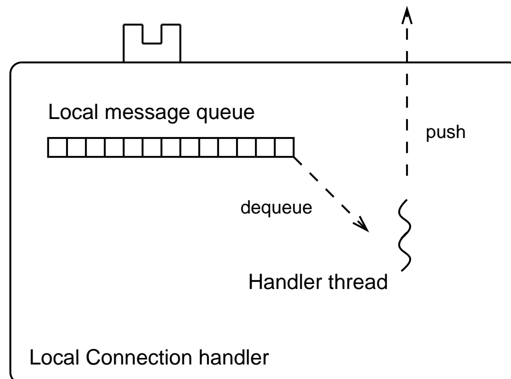

Figure 16: Local communication handler

This local communication is still handled by a special handler for transparency: this is a C++ subclass of the base connection handler, The MUX and ACTP running on top of this pseudo base connection have no idea that they are using local communication. This local handler is managed by the base connection management module. Whenever a call is done to find_base on the base manager interface with a local endpoint, the manager returns a reference to this connection instead of a base connection handler using an actual TLS/SSL connection.

A dedicated thread takes messages from the queue. This is done for transparency, because the layers on top of the connection handler do not know that the communication is in fact done on the local AOS. Otherwise when pushing a MUX open message over the local channel that triggers a reply these operations would be performed by the same (worker) thread, in which case we might run into problems with locking mutexes.

3.5.6 Flow Control of MUX & ACTP

As can be seen from figure 14, the base connection is the only active entity that handles the actual transfer of data over SSL. All incoming data is accepted and pushed into the corresponding buffers. For MUX connections, the incoming data queue kept in memory, for AC transfers, each packet is appended to the incoming AC (which is on disk).

Because MUX data is buffered in internal memory, the amount of data that can be buffered for MUX connections is finite. This means that whenever a remote AOS sends a large amount of data over the MUX channel, buffering all this data may occupy all internal memory available to AOS. If there is no control over the amount of MUX messages that can be buffered, this can cause AOS to run out of memory. A single remote AOS can probably use this limitation to mount a *Denial-of-Service* (DoS) attack.

To prevent this from happening, flow control has been implemented for MUX connections, which relies on TCP flow control. When the total sum of data enqueued in the MUX buffers exceeds the per-base limit for the amount of enqueued data, then the base connection receiver thread temporarily stops reading from the SSL connection. If the receiver thread stops reading from the SSL connection, the TCP layer will eventually stall the transfer. If sufficient space in the buffers becomes available again, the receiver thread can start reading again.

TCP flow control is needed because there are no semantics for flow control in the MUX (or ACTP) specification. The MUX protocol only has messages for opening and closing connections. Adding flow control to the MUX and ACTP protocols would require a redesign of the protocol, probably making it a lot more complex. The specification of the protocol is now quite minimal, which is fine for keeping everything simple, but it has its obvious flaws: if there is a slow receiver on a single MUX connection, then this can cause all other MUX connections that use the same base channel to block. Note that there is no flow control on the local base channel (which uses no TCP) in the current implementation, though this could be improved by imposing some limits on the size of the outgoing buffer of this local channel.

Flow control for ACTP packets has not been implemented, because these packets contain payload data which is immediately stored on disk. Thus the amount of occupied internal memory by ACTP packets is quite minimal. To prevent some remote AOS from sending a ridiculously large AC and occupying a large portion of the disk space, the AOS specification says that each AOS can have an internal limit for the size of an AC it can accept (if the AC is too large for the receiving AOS, it can simply remove the AC and discard further AC fragments immediately). The limit for this is not defined yet in the current implementation (effectively infinite) because the application domain can be practically anything. One can set this limit in AOS at compile time if required.

However, allowing ACTP and MUX to run over the same base connection has some implications, because flow control is not done for ACTP, where it *is* done for MUX connections. Should the incoming MUX buffers become too full, the base connection will be temporarily unable to receive *any* data, including ACTP packets. This is a little inconvenient, because ACTP transfers now also suffer from the MUX flow control, although they really should not have to be bothered by this (AC data is not buffered in memory but directly written to disk). So relying on TCP for MUX flow control is not ideal in this case.

Possible solutions to this problem exist, the simplest being to just disallow MUX and ACTP running over a single base connection. This is what the Java implementation currently does, and it at least means that ACTP transfers are not influenced by MUX flow control anymore. While this might use a little more AOS resources, this completely eliminates the problem of MUX interfering with ACTP. It would probably require only a small modification, so this might be worth adding in the C/C++ implementation, for example as an option that can be set at compile-time.

A better alternative is not to fall back on TCP flow control, but to add semantics for this to the MUX protocol. This is also better because it prevents the case where one virtual connection (for example a fast sender and a slow receiver) can completely fill up the queue of the base connection, effectively blocking the entire base connection.

3.6 Removing Unused Resources

There are two kinds of resources that can be automatically removed over the course of time, should they remain unused for a long time. These are the resources that cannot be explicitly removed by the AppMW processes: XIDs and base connections.

The AOS API specification does not include explicit removal of a single XID, but the specification describes the use of a timeout for these XIDs. If an ACTP transfer has not been initiated for a XID within a certain time after the XID was created, the XID is removed from

AOS, resulting in wait_ac with this XID returning an error. The AppMW level agent transfer will then have to be reinitialized (i.e. by calling prepare_wait_ac again, as shown in figure 15. The current implementation uses a timeout of 1 hour for this. If the ACTP transfer has already started (or completed), then the XID is not deleted after this timeout period. Furthermore, the AOS specification describes that invalidation of XIDs can happen when some errors occur during the ACTP transfer. Critical errors like I/O can cause a XID to be invalidated (so it can be removed), while noncritical errors (like a security suite intersection) do not invalidate the XID. A complete list of what should be done for each specific error is given in the AOS specification. A special collector for the cleanup of these XIDs is scheduled every minute, which checks all of the registered XIDs to see if any of these can be removed due to invalidation.

The other kind of resources that cannot be cleaned up by middleware processes are base connections. Whenever a base connection is unused it can be eventually removed. There are different policies for this, but the current implementation leaves the base connection open for a few (currently 5) seconds before removing the base connection. Better policies exist however, for example cleaning up base connections only when AOS runs out of file descriptors for TCP sockets. This is a little difficult to accomplish however, because new TCP connections can also come in at the base connection listener, in which case a connection cannot always be accepted. Similarly, new TCP sockets can be requested for incoming connections on the SunRPC dispatcher. These problems have to be carefully analyzed to allow AOS to behave correctly in all cases when it runs out of file descriptors. The simplest implementation is currently used: close the base connection almost immediately after it is unreferenced. If a new connection to the endpoint is requested again after 5 seconds, then a new base connection has to be created. SSL session caching (not implemented) could speed up this process.

3.7 Cookie Crash Persistence

(currently only partly implemented)

A cookie and its associated resources can partly survive crashes. This is useful because AOS runs as a separate process, with possibly many middleware processes running on top of it. The problem with supporting crash persistence is that all processes using AOS should be at least aware of AOS crash recovery. It might be difficult to handle a loss of resources for AppMW layers without any recovery mechanism in AOS.

Should AOS crash, it should be possible to recover at least the cookies and some resources that were registered in AOS. Currently a simple implementation is used for this, which keeps cookies, finalized ACs, and XIDs for which an AC already has arrived available for middleware processes using AOS. Other resources, such as listen endpoints, open connections and running ACTP transfers/XIDs do not survive a crash of the running AOS.

To let certain resources survive AOS crashes, it is necessary the this information is stored in persistent storage (on disk) rather than just keeping this information in memory. Finalized ACs can be easily recovered, because their last finalized or arrived state is always present in a well-known place on disk (see section 3.4.1 on AC finalizing). If such a file can be found for an AC, then it can be recovered.

Recovering the cookie table can be done by storing each cookie entry in a well-known location on disk. A snapshot of the cookie table entry is stored on disk whenever an AC is

added to the table entry, or a child role was created for this entry. Endpoints and connections associated with a cookie are not stored on disk, as these do not survive an AOS crash by specification. The cookie table entry is stored on disk in the AOS temporary directory, under the file name roles/<role-ID>. The role ID is an identifier which is used internally to AOS to index cookies and child cookies. Recovery starts with recovering the init role (which has a well known ID) and the cookie information is rebuilt by recovering all child cookies and resources recursively.

XIDs are a little tougher: only a completed transfer can be recovered. This can be done because the XID and cookie of the owner are also stored in a well-known location on disk. The cookie of the owner is stored in agentcontainers/ac-ID/owner and the XID which was used to receive this AC is stored in the file agentcontainers/ac-ID/xid. This approach is necessary because the AC ID is not known to the client process until it calls wait_ac with the corresponding XID.

Should AOS crash, the client process receives an error indicating AOS was reset when it tries to use its cookie for calling AOS. The client process has to call the reenable_role with this cookie to let AOS know that it would like to continue using AOS with its cookie. Should the role not be enabled within a certain time span (e.g. a week) then the cookie is deleted by AOS.

At the time of writing crash persistence is not completely implemented, though most of the functionality (role state saving, AC recovery) is already present. The only thing left for adding crash persistence support to the AOS C/C++ implementation is recovery of the role table and ACs and testing. This might not prove to be that much work, so the final version of AOS (for this project at least) might even have a complete implementation of crash persistence.

4 Results & Testing

This chapter presents the results of the testing that was done for the C/C++ kernel implementation. There are some simple test programs in the source distribution, aimed at testing the behavior and performance of the kernel. Testing the correctness of the AOS implementation is especially important for the Java implementation and the C/C++ implementation to communicate, for example when running ACTP between different implementations.

The main focus of this chapter is on performance testing. Both kernel implementations are compared to see if there are large performance differences between them, but these tests can also highlight inefficiencies in any one of the implementations, which could be useful for future speed optimizations.

These tests have been done to provide insight in optimizations that can be made to provide better performance. The tests are not meant to be used to immediately tune the performance of the kernel implementations (this can always be done later). The tests are not exhaustive, but rather these tests give an overview of the main differences between both implementations and highlight the most important performance characteristics.

4.1 Test Setup

The specifications of the machine used for running the tests was:

CPU/Memory:	AMD Athlon XP 2500+ (1.83Ghz), 512 MB RAM
Storage:	Samsung SP1614N 7200RPM/8MB (ATA) drive, using ext3 file system
OS:	Mandrake Linux (2.6.3-4mdk kernel)

The compiler used for the C++ kernel was g++/gcc from the GNU Compiler Collection 3.3.2. The Java kernel was compiled and tested using the Sun 1.5.0_04 JDK with standard JVM configuration, with Just-in-time (JIT) compilation enabled. JVM heap management, garbage collection and JIT settings of the JVM might well influence its performance in some cases. In particular, when the Java kernel has to allocate a lot of memory or creates a large number of objects, heap and garbage collector settings may influence the kernel's performance significantly. JIT compiler settings may also improve performance in some areas, in particular when cryptography has to be done. The cases where these issues may influence the results are described in the test sections in this chapter. We did not attempt to optimize the settings of the JVM for the tests described in this chapter.

The machine was a standalone machine so there were only a few processes running in the background, but none that used many resources, so measurements are not affected by external sources.

4.2 Performance Tests

This section presents the results and observations of the performance tests that were done on both kernel implementations. All tests were written in C and used the SunRPC client stub provided with the C/C++ AOS kernel implementation. With this SunRPC client stub the test program can contact both kernel implementations, so the tests for both AOS implementations only had to be written once.

Three main test series were done:

- Dispatcher performance: latency and throughput (section 4.2.1)
- MUX connection performance: setup time, latency and throughput (section 4.2.2)
- AC finalize performance (section 4.2.3)

4.2.1 Dispatcher Performance

All middleware processes use a dispatcher as their main RPC mechanism to contact AOS. RPC requests are probably more expensive than native function calls because the RPC request has to travel over an IPC channel and the function arguments and return values need to be serialized. To see what the impact of such a dispatcher is, this section contains test results which measure the performance penalty the dispatcher introduces. Measured are latency (instead of a local function call, the call must travel through the IP loopback device) and the throughput (because the data has to be sent over a local TCP connection) of the dispatcher.

Latency is measured by sending small function calls (segment read and write) with varying amounts of payload data to the dispatcher. This test was repeated 1000 times and the average and standard deviation of all the individual times were calculated. The latencies of function calls with more payload data were also measured to provide insight in the maximum data throughput that is possible using the SunRPC dispatchers.

In order to make sure that the SunRPC dispatcher is isolated so that AOS does not do any extra work, the implementation of these function calls was modified in both kernel implementations. For a segment write, the function was modified to return immediately, and for a segment write, the function immediately returns with an empty buffer of the requested size (so no actual read and write calls are executed on the segments). Both implementations use a dedicated thread to perform the result, but apart from shipping data no more work (like cookie lookup or segment access) is performed by AOS.

The performance numbers are depicted in table 1a and 1b, where the first number in each cell is the average time that was needed to complete a single function call, and the number between brackets represents the standard deviation for all 1000 runs. From these latencies also the throughput can be derived (shown in figure 17 and 18)

	1 B	16 KB	32 KB	64 KB
C++ write	0.24 (2.25)	0.33 (2.18)	39.99 (0.05)	0.42 (0.67)
C++ read	0.24 (2.26)	0.31 (2.23)	40.23 (2.37)	0.81 (4.18)
Java write	3.97 (14.02)	4.06 (13.28)	41.42 (7.03)	4.29 (12.30)
Java read	2.26 (8.51)	2.48 (8.54)	2.77 (8.51)	3.25 (9.12)

Table 1a: Latency times in milliseconds (average + standard deviation)

	128 KB	256 KB	512 KB	1024 KB
C++ write	0.84 (3.13)	1.51 (2.14)	3.24 (2.31)	6.44 (2.13)
C++ read	0.79 (2.01)	1.67 (2.11)	3.21 (2.20)	5.86 (2.33)
Java write	5.39 (12.92)	8.45 (15.35)	80.38 (5.73)	87.92 (7.52)
Java read	4.09 (9.76)	6.81 (13.43)	12.67 (19.01)	83.76 (2.96)

Table 1b: Latency times in milliseconds (average + standard deviation) continued

In table 1, the segment read and writes for both implementations can be observed. The number above each columns denotes payload size that was sent and received by each of the 1000 calls to the read and write functions, e.g. 128 KB means the times were obtained by performing 1000 calls to the dummy seg_write and seg_read implementation with 128 KB of payload data. The contents of the buffers are not used (discarded immediately) so that pure RPC dispatcher overhead is measured.

What can be seen from table 1 is that the standard deviation is quite large, usually around 2 milliseconds for the C/C++ implementation, independent of the amount of payload data (this can be caused by scheduling of the worker thread for the job). Interesting behavior can be observed when writing 32 KB of data to the dispatcher, which is very slow for both kernel implementations. This means that there is likely to be a problem with the client-side stub of the C implementation, as the performance of the write test with 32 KB to any of the kernels is a lot slower than it should be, and read_seg shows the Java kernel to be faster than the C/C++ kernel in this 32 KB range.

When receiving data (read_seg) from the AOS kernel (in which case the AOS kernel writes the data back to the client) there are still performance problems for the 32 KB test using the C/C++ kernel, but these problems are absent from the Java AOS. This is further evidence that there is a problem with the SunRPC write routines used for the C implementation (which are in libc on Linux), as this problem does not exist having the Java kernel sends the data (which uses different routines for writing).

What causes the sending of 32 KB data as a payload to be so slow over SunRPC will require some investigation of the library that implements these SunRPC write routines, as it is probably not an AOS internal problem, but exists in all function calls that use the libc routines for writing data.

Some other tests (results not shown) showed that the weak spot of the SunRPC dispatcher lies between 20 KB and 40 KB: any amount of data between these sizes is sent very slow.

Apart from the poor performance in the 32 KB tests, the results scale about linear (see also throughput graphs in figure 17 and 18). However, the Java AOS seems to have some problems when reading 512 KB data or more, or when writing 1024 KB or more. It is difficult to say what exactly causes this increase in latency, but it could very well be caused by the garbage collector reclaiming memory.

To compare the throughput of both dispatchers, see figure 17 and 18. These graphs contain the throughput of both the dispatchers, the results in these graphs are derived from the results of the latency tests in table 1.

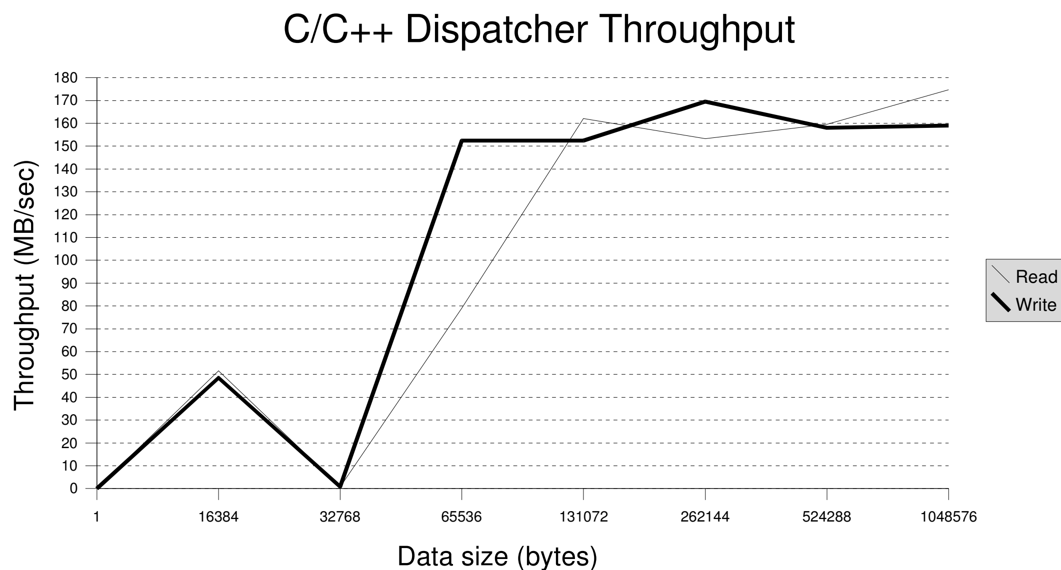

Figure 17: Average dispatcher throughput (C++ kernel)

Figure 17 contains the throughput of the dispatcher of the C/C++ AOS when reading and writing varying amounts of payload data. The throughput is somewhat limited when using small amounts of payload data per RPC call, but from 64 KB and higher, these values are hardly dominated by latency anymore.

The throughput of sending SunRPC requests over a local TCP connection can reach values as high as 150-170 MB/sec for the C/C++ implementation. Whether this is enough depends on what AOS is supposed to do with the data: if the data is sent over a relatively slow (encrypted) connection or written to disk, then this 170 MB/sec is probably adequate and will not be a large bottleneck for the overall performance.

The next figure contains a similar graph of the dispatcher throughput for the Java AOS.

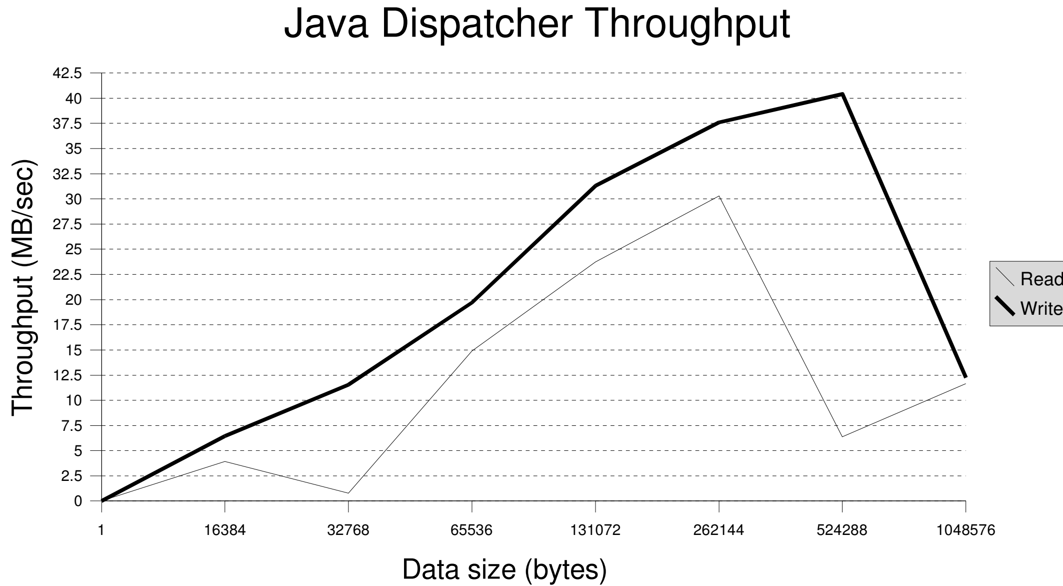

Figure 18: Average dispatcher throughput (Java kernel)

The Java AOS dispatcher can not reach the throughput of the C/C++ kernel implementation, it is at best 4 times as slow (40 MB/sec), except for the 32 KB region which is not problematic when reading from the Java AOS.

When sending more than 512 KB of data the performance of the Java kernel decreases. This can be caused by the garbage collector, because all data that is transferred in each one of these tests is immediately discarded. Because freeing memory is a manual operation in C/C++ but an automatic operation in Java, it is likely that there is more overhead when having the garbage collector take care of this.

In conclusion, the C/C++ implementation has the faster dispatcher. The latency for a small function call is a lot lower than for the Java implementation (0.24 ms compared to 2.26ms, see table 1a). The maximum throughput that can be achieved is about 4 times higher (170 MB/sec for C/C++ and only around 40 MB/sec for SunRPC in Java).

4.2.2 Virtual Connection Performance

This section contains the results of the performance tests for the AOS communication channels. AOS offers multiplexed communication channels with a configurable level of encryption, of which the latency and throughput were measured. Inter-AOS communication channels use SSL over TCP, but for each middleware process there is also a dispatcher over which the data has to be transferred to the middleware process. The measurements that were done are:

- Latency for connection setup and ping latency (including dispatcher)
- Throughput of AOS to AOS channel (without dispatcher)
- Throughput of middleware to middleware channel (including dispatcher)

These tests were run for both kernel implementations using two different cipher suites: a fast RC4 stream cipher with MD5 message digest, and the computationally intensive triple-DES cipher using SHA1 message digest.

First we will look at the AppMW latency times for connection setup. MUX connection setup requires the setup of a base connection if a suitable base connection does not yet exist. If such a base connection already exists, then this connection can be reused. To see what the difference in MUX connect times is, the latencies for both types of MUX connection setup are measured. This will probably show a difference in setup times: when reusing an existing base connection there is no need to perform an expensive RSA key exchange. This test can help in showing the benefit of multiplexing connections over a single SSL connection.

The test program executed 10 AOS connect function calls, in which case the first connect call implicates the RSA key exchange, but the later connects do not. This test was repeated 10 times, and the average values are shown in the next table. The column *connect over new base* contains the average value for a connect calls that included RSA key exchange, and the normal *connect* is for the cases in which AOS could reuse a base channel.

The next table also contains AppMW to AppMW latencies, which were measured by a client program sending 1 byte to a server program. As soon as the server receives a byte, it sends one byte back as a reply. The client measures the time between the start of sending this one byte and the receiving of the reply as the complete end-to-end latency of AOS. This test was repeated 1000 times and the average value is also shown in the next table under the column *ping*. The test used the RC4 cipher, but the exact cipher that is used does not really matter because all of the AOS supported suites use an RSA key exchange.

	1) connect over new base	**2) connect**	**3) ping**
C/C++	68.12 (3.13)	0.52 (0.35)	1.95 (10.28)
Java	195.45 (75.32)	22.12 (32.17)	21.49 (27.73)

Table 2: Latency times (ms) for virtual connection operations for (1) MUX connect including new base connection, (2) MUX connect over an existing base connection and (3) ping over a MUX connection. Values are the averages of 10 runs (connect) and 100 runs (ping), the standard deviation is between brackets

What can be seen from this table is the large difference between the different connection setup times. As could be expected, the RSA key exchange has a large impact on the total connection setup time: both the C/C++ and Java implementations take a lot longer to make a connection when this requires an extra base connection setup. This is a good reason to use connection multiplexing. The ping times for the Java AOS are somewhat slow compared to the connection setup time (20 ms), but the ping time for the C/C++ kernel is also slower than a connection setup. This is because the ping data is sent over the dispatcher of both AOS implementations, whereas the connection setup is AOS to AOS only (the connection is already completely created inside AOS, but the middleware has not accepted it yet).

Next will be some measurements of the AOS to AOS channels. We will try to see what the maximum throughput of the base and MUX channels are, which is done by creating MUX connections and sending data over it. To also acquire insight of the impact of multiplexing the tests were done with a varying amount of virtual connections. To make sure the dispatcher is not influencing the results, the tests are first done directly on top of the native AOS API, which required inserting code directly in the AOS code. The test for native AOS throughput was therefore only done on the C/C++ implementations.

The test sends data using a varying number of virtual connections and each virtual connection sends an equal amount of data. The test is performed by having a sender thread for each MUX connection on one AOS instance, and a receiver thread for each connection on the other AOS instance. As with all tests, both AOS instances run on the same host. Each sender sends its amount of data, and then waits for the receiver to send back a notify. The receiver sends back a notification only after it has completely read all data. The sender can then use the time from when it started sending data to the time it received the notification to determine the total throughput of the operation.

In table 3 we see the results for the MUX throughput tests directly on the native API. In all tests 10 MB was transferred between two AOS instances, where the column label indicates the number of virtual connections that were used for the transfer. For example, in the column labeled '5' the data was equally divided over 5 virtual connections, so each one of them sends 2 MB of the total. Running this test directly on the native API will probably give a good insight of the maximum raw throughput that is available. The test was done twice, using RC4 and triple-DES encryption. All MUX tests were done 5 times, the averages are in the tables.

The first number in each cell represents the total utilization of the channel, and the numbers between the brackets give the bandwidth of the channel that each client itself sees. For example, when using 2 senders to transfer 50 MB of data, both send 25 MB. Suppose that sender 1 completes in 1 second, and sender 2 completes in 2 seconds, then the total operation takes 2 seconds, meaning that the total utilization of the channel is 25 MB/sec. Sender 1 had a transfer rate of 25 MB/sec, but sender 2 only had a transfer rate of 12.5 MB/sec. This leads to an average of 18.75 MB/sec for all senders.

The first number in each cell represents the total utilization of the channel, and the numbers between the brackets give the bandwidth of the channel that each client itself sees. For example, when using 2 senders to transfer 10 MB of data (using RC4) sender 1 completes in 0.35 seconds (14.3 MB/sec) and sender 2 completes in 0.46 seconds (10.9 MB/sec). The average of the bandwidth is then 12.6 MB/sec, but because the total operation (10 MB) took 0.46 seconds, the utilization of the channel is 21.8 MB/sec. 2x12.6 MB/sec is not the same as 21.8 MB/sec, because the last 0.11 seconds of the complete operation sender 1 was already finished: the total utilization of the channel is based on the slowest sender.

Number of virtual connections

Cipher		1	2	5	10
	RC4	23.6	21.8 (12.6)	18.9 (5.3)	18.8 (2.3)
	3DES	4.8	4.4 (2.9)	3.1 (1)	1.9 (0.5)

Table 3: Total & average (per-connection) throughput (MB/sec) for sending 10 MB of data without dispatcher (C/C++ kernel). The first numbers contains the total throughput of the entire operation, the number between brackets is the average times for each virtual connection.

Table 3 shows that using RC4 is much faster than 3DES encryption. Also, the total utilization of the base channel declines when more threads are sending their data at the same time: the 18.8 MB/sec for RC4 with 10 senders is a lot slower than the maximum of the channel (23.6 MB/sec). This can be caused by the extra thread overhead, more so because this test runs 2 AOS instances on the same host machine. Also, the receiving AOS must also handle all incoming data for all receivers, after which each receiver individually sends out a notification. The average value (2.3 MB/sec for 10 senders) indicates that the average transfer rates are still quite high, but some receivers can be just a little slower than the rest.

The next two tables show the results for the MUX performance tests on the C/C++ kernel, this time with a dispatcher. This means that all data has to be sent *two* extra times over the dispatcher (once for sending and once for receiving). Because both AOS instances run on the same host, all instances use the IP loopback device for any communication.

Number of virtual connections

Data set size (MB)		1	2	5	10
	10	16.3	15.1 (10.9)	13.6 (4.6)	3.2 (1.9)
	25	17.6	16.8 (11.5)	15.8 (4.5)	7.4 (2.6)
	50	17.8	17.8 (9.3)	16.8 (4.3)	12.9 (2.9)

Table 4: Total & average (per-connection) throughput (MB/sec) using RC4-MD5 **(C/C++)**

Table 4 shows that the total utilization of the RC4 channel suffers in comparison to the native tests from table 3. This can be a caused by the use of the loopback device for both AOS instances (although it should be able to reach speeds of over 150 MB/sec according to figure 17) but the extra work that needs to be done for encryption probably limits the overall throughput for this device. More realistic tests should use AOS instances running on multiple machines connected by a fast network.

When using 10 senders it can be clearly observed from table 4 that the performance is a lot lower than when using only 5 senders, especially for the 10 MB test, but with the larger tests the performance gets better. Even an 100 MB test for 10 senders (results not shown) showed a total utilization of the channel of more than 17 MB/sec, which is already near the maximum that can be reached when using RC4 over the SunRPC dispatcher. Still the 3.2 MB/sec for 10 MB using 10 senders is significantly slower than when using fewer senders, or a larger data

set for testing. The 10 MB test is probably too small to give accurate values, because when even a single thread is a little slower than the rest this has a large impact on the total transfer rates. For some reason, a few threads were always delayed when using 10 or more senders, this could be an issue internal to AOS, and it might be worth investigating what causes this difference between the threads.

As a small extra comparison, a test was done without encryption, using the NULL-MD5 cipher. This cipher uses no encryption, but only a MD5 message digest (results not shown). Over the native AOS API it was possible to achieve speeds of more than 35 MB/sec, but when using the dispatcher this was usually around 21 MB/sec. Clearly the dispatcher limits the performance of the base channel for fast ciphers.

		Number of virtual connections			
		1	2	5	10
Data set size (MB)	10	5.3	5.2 (2.7)	4.7 (1.5)	2.9 (0.9)
	25	5.4	5.3 (2.7)	5.1 (1.4)	5.1 (0.9)
	50	5.4	5.3 (2.7)	5.2 (1.4)	5.1 (0.8)

Table 5: total & average (per-connection) throughput (MB/sec) using 3DES-SHA **(C/C++)**

Table 5 shows that the performance using 3DES is quite consistent, even when using many senders the total utilization of the channel is hardly affected. Similar to the results of the RC4 tests from table 4, larger data sets show better performance in all tests, but the differences are not that large here, probably because the test is more limited by the slower 3DES encryption. For the 3DES tests also bad performance can be observed for the 10 MB data set with 10 senders. There could be some internal issues in the C/C++ AOS implementation (maybe mutex locking for internal synchronization, or thread scheduling) that limits the performance of some of the threads. This might be worth investigating.

Interestingly enough, 3DES encryption reaches speeds over 5 MB/sec, which was not even possible when running over the native AOS API. What really causes these tests to be faster than their native counterparts is unknown, but all tests were repeated multiple times with similar results. Clearly, the dispatcher does not limit the performance of the base channels when using a computationally intensive cipher like 3DES. It is strange that the tests running over the SunRPC dispatcher are even faster than the 3DES tests that were done on the native API (table 3). What issue there was with the test program is not known, but it had no bad effect on the RC4 throughput so this is not easily explained.

Next we will look at how fast the Java implementation can perform cryptography for comparison. Unfortunately no tests were done directly on top of the native Java API, which would require some new code to be written in Java. The test program is not that complex so this might not be too hard to do later if desired. Tables 6 and 7 show the performance of of the test program running over the Java AOS for RC4 and 3DES.

		Number of virtual connections			
		1	2	5	10
Data set size (MB)	10	3.0	4.1 (2.1)	3.8 (0.8)	3.4 (0.4)
	25	3.3	4.3 (2.2)	4.1 (0.9)	3.3 (0.4)
	50	3.4	4.5 (2.3)	4.1 (0.8)	3.7 (0.4)

Table 6: Total & average (per-connection) throughput (MB/sec) using RC4-MD5 **(JAVA)**

It can be seen from table 6 that Java is a lot slower when doing cryptography: RC4 is at best 4 times as slow when done in Java than it is when done in C. The total utilization of the channel is usually around 4.5 MB/sec, though when using only one sender the performance is a little lower, typically around 3.5 MB/sec. Probably there is a more efficient use of resources when having more than one sender (e.g. interleaving of senders lead to better usage of all communication channels). For 5 and more senders, the Java kernel is slower again compared to 2 senders, a similar behavior to what the C/C++ AOS shows. Table 7 shows the results for the Java implementation when performing 3DES encryption for the communication channels.

		Number of virtual connections			
		1	2	5	10
Data set size (MB)	0	0.64	0.76 (0.38)	0.70 (0.14)	0.70 (0.07)
	25	0.76	0.72 (0.36)	0.69 (0.14)	0.69 (0.07)
	50	0.77	0.76 (0.38)	0.68 (0.14)	0.68 (0.07)

Table 7: Total & average (per-connection) throughput (MB/sec) using 3DES-SHA **(JAVA)**

Table 7 shows that 3DES in Java is slow: only a maximum of 750 KB/sec is possible. The total throughput results are consistent overall, meaning that the performance of the virtual channels is not limited by the amount of senders or the presence of a dispatcher, but rather the performance is influenced most by the relative slow speeds of performing 3DES in Java.

As a conclusion, the presence of a dispatcher only limits the performance of the really fast ciphers, but a test setup using multiple machines connected by a network could prove to show more realistic numbers, because the base connections might run over slower channels so the choice of cipher might not limit the total performance that much. For slow encryption (like 3DES), the dispatcher throughput is no limitation to the entire process. When performance is desired, the C/C++ is clearly the better choice.

4.2.3 AC Finalize Performance

The next series of tests measure the performance of the AC finalize operation. This is an expensive operation because it relies on cryptography (hashing for creating the segment checksums and public key cryptography for creating the signature) after which all segments must be (compressed and) stored in a ZIP archive. This section describes the performance of the finalize operation for both kernel implementations, for various AC and segment sizes.

Measurements of the time to ship an AC over the network (ship_ac and wait_ac) were not performed, as the connection throughput was already determined in the MUX connection tests in section 4.2.2. ACTP transfer times are expected to be similar.

Measuring the finalize operation itself is quite simple: once all the segments are created in the AC, a process can call the AOS finalize function which blocks until the operation is done. These tests were done for varying AC sizes, ranging from 0.1 MB to 100 MB. Each individual test creates a number of segments with the exact same content and size. Once this is done the time for the finalize call to complete is measured. All tests are done 5 times, the average and standard deviation are derived from the measurements. Both AOS implementations used the fastest ZIP compression.

For each individual test the segments contain the same data, this data is read from a file and stored in the segments. For example, when creating 100 segments of 100 KB each, each segment will contain the first 100 KB of the test file as its data. Because compression can be done on the AC during finalize, there are two different data sets used for the segment contents: one contains data from an MP3 file (which is hardly compressible) and one contains data from an executable binary program (about 50% compression possible).

The C/C++ kernel uses a memory cache of 256 MB by default, though segments larger than 1 MB are never cached in memory, but they are stored directly on disk. The Java AOS has no memory cache, and it keeps all segments directly on disk. Comparing these different strategies is not entirely fair, so tests with a disabled memory cache for the C/C++ AOS are also performed. The timing results for the default configuration of both AOS implementations (MP3 data set) are shown in the next table.

The C/C++ vs Java column compares the results of both implementations. The numbers in this column represent the percentage of time that the C/C++ kernel needed to perform the operation relative to the Java implementation. For example, test 2 is performed in 0.12 seconds on the C/C++ implementation, this is 42.86% of the time that the Java kernel needed for this test. The lower these numbers are, the faster the C/C++ kernel runs in comparison to the Java AOS implementation.

test	*#segs*	*segsize (KB)*	*Finalize times (sec)*				*C/C++ vs Java (%)*
			C/C++		*Java*		
			avg	*dev*	*avg*	*dev*	
1	10	10	0.03	0.01	0.04	0.00	**75.00**
2		100	0.12	0.00	0.28	0.04	**42.86**
3		1000	1.49	0.04	3.27	0.12	**45.57**
4	100	10	0.05	0.01	0.19	0.05	**26.32**
5		100	1.27	0.06	2.69	0.21	**47.21**
6		1000	13.14	0.45	27.69	1.82	**47.45**
7	1000	10	0.38	0.01	1.66	0.03	**22.89**
8		100	10.57	0.63	22.66	0.21	**46.65**

Table 8: Finalize times in seconds (averages + standard deviation) for the tests using the non-compressible dataset (MP3 data). All tests were run 5 times for both kernel implementations.

The C/C++ implementation is usually more than twice as fast as the Java AOS, except for a few cases the C/C++ implementation only needs between 45%-50% of the time that the Java AOS needed for the finalize operation. Except for the differences in speed, the behavior of both kernel implementations is nearly similar. The two tests that work with a 100 MB AC (test 6 and 8) are the slowest on both implementations, but the difference is still around 47%. It is interesting to see that both implementations perform the 100 MB AC test faster with 1000 segments of 100 KB (test 8) than with 100 segments of 1 MB (test 6). Overall, when speed is demanded, the C/C++ is the better choice. The results for finalize for a compressible data set are shown in the next table.

test	*#segs*	*segsize (KB)*	*Finalize times (sec)*				*C/C++ vs Java (%)*
			C/C++		*Java*		
			avg	*dev*	*avg*	*dev*	
1	10	10	0.03	0.00	0.05	0.01	**60.00**
2		100	0.10	0.01	0.24	0.03	**41.67**
3		1000	0.80	0.02	1.95	0.09	**41.03**
4	100	10	0.11	0.00	0.33	0.05	**33.33**
5		100	0.88	0.01	2.04	0.09	**43.14**
6		1000	8.19	0.87	17.68	0.79	**46.32**
7	1000	10	1.08	0.13	2.86	0.07	**37.76**
8		100	8.29	0.28	18.83	0.62	**44.03**

Table 9: Finalize times in seconds (averages + standard deviation) for the tests using the compressible dataset (an executable binary). All tests were run 5 times for both kernel implementations.

Table 9 shows that when using compressible data, the finalize times become lower for both AOS implementations While the C/C++ implementation has a little more benefit from this data set, the relative differences between both are about similar to what can be seen from table 8. Especially in the 100 MB tests, the compressible AC is created much faster than the non compressible AC, though this time there is hardly any speed difference between tests 6 and 8 which are about equally fast (in the test with MP3 data there was a considerable difference between both tests). This can be due to the decrease in the amount of disk access that needs to be done for writing the result (the resulting AC is around 45 MB where it was 100 MB for the MP3 data set), or a more efficient execution of the ZIP compression.

To see what the impact of the disk cache is for the C/C++ implementation, the two tests with the largest ACs (tests 6 and 8, 100 MB AC) were repeated on the C/C++ AOS with a disabled memory cache. This can give more insight in how far I/O times (for segment read and write) are limiting the finalize times for both implementations. The results of these tests are shown in table 10, for both the compressible data set (6c and 8c) and the non compressible data set (6nc and 8nc).

test	*#segs*	*segsize (KB)*	*Finalize times (sec)*				*C/C++ vs Java (%)*
			C/C++		*Java*		
			avg	*dev*	*avg*	*dev*	
6 nc	100	1000	14.26	0.61	27.69	1.82	**51.5**
6 c	100	1000	7.84	0.21	17.68	0.79	**44.34**
8 nc	1000	100	18.02	5.40	22.66	0.21	**79.52**
8 c	1000	100	13.90	6.92	18.83	0.62	**73.82**

Table 10: Finalize times in seconds (averages + standard deviation) for the tests using the non-compressible dataset (an executable binary). All tests were run 5 times for both kernel implementations.

Table 10 shows that relying on disk for segment store is bad for the performance of the C/C++ implementation. This is not really noticeable when using 100 segments of 1MB for either data set, but when using 1000 segments (tests 8c and 8nc) the performance becomes significantly worse. Not only is the finalize slower, but also the standard deviation is quite large: test 8 with non-compressible data has a standard deviation of 5.40 which means that sometimes the C/C++ implementation is even slower than the Java implementation. It is a little strange why this behavior can not be observed in the Java implementation (which shows only little variation) but this might be a problem internal to the C/C++ AOS implementation: disk caching was not thoroughly tuned for performance.

Strange results can be seen in test 6c, which is on average faster than when using the internal memory cache. How this exactly is possible is unknown, but it may be that the file system cache has no problem containing all the segments and therefore the memory cache operations become redundant. The total operation then probably becomes faster because the segments do not need to be moved to disk, though this is not a complete proof. Keeping large segments (test 6) on disk does not decrease the performance a lot, although this might change when more AppMW process are concurrently finalizing. Having many small segments on disk is much slower, possibly because the file system can not handle these as efficiently as a few larger segments. If only a few (large) segments are used, the performance is not very bad compared to using a memory cache for the segments.

To get a better insight into where the most time is spent internally in the entire finalize call some extra measure points were inserted in the AOS code (for the C++ kernel only). The results for a typical run of test 6 (both data sets) are shown in the next table.

Action	Time (seconds)	
	Non-compressible data	**Compressible data**
Checksumming segments	0.69	0.59
Signing TOC	0.04	0.02
ZIP file creation	11.58	7.50
Total time	**12.31**	**8.11**

Table 11: Detailed benchmarks of AC finalizing steps

Interestingly enough, table 11 shows that the creation of the ZIP file dominates the total finalizing time. The cryptography (creating a SHA1 hash for all the checksum and creation of the signature) only takes a small percentage of the total time. Although if the memory cache is disabled the checksum creation time increases (measurements not shown) because all segments checksums can not be directly created from memory.

Creation of a ZIP file consists of compressing the data and storing the resulting file on disk. The ZipArchive library also flushes this file to disk, so the times for this could not be measured separately. By default, both AOS implementations use level '1' (best speed) for the compression of the AC. As show in table 11, this compression works faster if the data can actually be compressed. Using the compressible data set, this results in a 45 MB archive, while otherwise this would result in 100 MB file.

To see whether this difference in ZIP archive create time is caused by compression of the data, or by I/O overhead (synchronizing the file to disk) some measurements are also done with compression level zero (store only, no compression). The results are shown in table 12.

Action	Time (seconds)
Checksumming segments	0.58
Signing TOC	0.02
ZIP file creation	2.62
Total time	**3.22**

Table 12: Detailed benchmarks of AC finalizing (compression level 0)

Table 12 shows that without compression, the finalize times are significantly reduced. The only problem with this approach is that for test 6 this always results in a 100 MB finalized Agent Container. Whether the finalize time or the AC size is more important depends on what is done next: if the AC is to be transported over ACTP over a relatively slow link then compression is probably a good thing to do, but if the data can not be compressed there will be hardly any difference in size, even though then the finalize times are 4 times as slow (3.22 versus 12.31 seconds).

If the data can not be compressed, the creation of a ZIP file takes a lot more time than when the data can actually be compressed. This is probably caused by the ZIP library routines, which apparently run more efficient if data can be compressed. When the data can be compressed, ZIP level 0 is still more than twice as fast as level 1, but compression results in an AC of only 45 MB, whereas otherwise the AC is always 100 MB regardless of its

contents. Unfortunately, AOS cannot guess the level of compression that is best suited for each individual segment so the current AOS C/C++ implementation uses a default value for all segments. The compression level can be changed at AOS compile time if desired.

Even though segments are typed, AOS cannot guess what kind of compression is best suited for each segment. To make this possible, either something similar to MIME types, or manual inspection of the contents (like the UNIX "file" utility) have to be done. In these cases, AOS could make a difference between the compression of different segments, but such a mechanism is not present in the current AOS implementation. If it is possible, then setting the compression level to zero for MP3 or other non compressible data can give extra performance, while hardly increasing the total finalized AC size.

Note that the results above may differ when a different file system is used. The current setup uses a Linux ext3 file system, which uses journaling resulting in fast write/sync times. Using a more traditional UNIX file system may result in larger I/O times, possibly even tipping the scale to benefit from compression in most cases. More extensive testing is required to test the influence of this and AC transfer times, to see whether AC ZIP file compression is useful.

4.3 Correctness Testing

Correctness testing has been done to ensure that the AOS implementation behaves as it should. The term correctness covers various tests, but the most important one is that all AOS function calls should return exactly the data (and error codes) as should be expected. This means that as a bare minimum, all read and write operations should return the correct values, data stored in segments and sent over connections should be as expected, and no data access should be possible to resources without the corresponding cookie, etc.

More difficult to test are probably the cases that lie inside the program code. For example, it can not be observed from a middleware process whether the base connection actually uses the correct security suites for the base channel, if base channel reuse always finds a base channel if there is one, if the finalized AC contains the correct data, etc. These tests must generally be done by hand, or by inspecting the log files of AOS, and as such these are a little harder to test automatically.

Other important correctness tests are about the cooperation of both AOS implementations. Connections must be able to run between both kernel implementations, as well as AC transfers. The arrived AC should be correctly interpreted and both AOS implementations should be able to interpret the ACTP reply (either an error or a receipt) and react accordingly. These tests are a little easier to perform again, because this time the middleware can easily check the contents of the data: it does not need to know what AOS implementation it is actually using.

There were already some testing programs discussed in section 4.2, and even though these tests were mostly aimed at testing performance, the results of these tests should also be correct. As these tests really test AOS under high load, adding some (usually) trivial checks to the test programs can also make them suitable for correctness testing. For example, the data sent out by the MUX tests should be the same as the data that is received. These tests can also be done by having different AOS implementation connect to each other.

Correctness testing is still difficult, especially for testing under high load for finding race conditions. Race conditions usually happen only in specific situations, which is a drawback

of programming with threads: because the program becomes nondeterministic a certain error can only occur in certain specific cases. If a program bug is hard to reproduce, then it is generally even harder to solve. The concurrency tests that have been done are tests for testing MUX performance (see the virtual connection tests in section 4.2.2) and concurrent AC finalize tests: having 10 threads simultaneously finalize an AC should result in the same finalized Agent Containers as when this finalizing is done one at a time. This does not guarantee that the AOS implementation is completely free of bugs, but an indication that most concurrency is handled well in the C/C++ implementation. When writing a testing program, it is generally best to always run the AOS instance in a debugger (like gdb), and with logging enabled in AOS (so you can trace all AOS steps), so that in the case that something should happen, there is a better chance of finding out what exactly has happened.

5 Conclusion

One of the main motivations for implementing the Agent Operating System in C/C++ was performance. The performance tests for the individual AOS components show that in nearly all cases the performance of the C/C++ implementation is considerably faster than the Java implementation. Achieving this higher performance was mostly obtained by this difference in programming language, as an interpreted language like Java simply suffers from an extra translation round. Large computations take more time this way, but Java also suffers from higher latencies as shown in the tests in chapter on dispatcher latency (section 4.2.1) and virtual connection latencies (section 4.2.2). However, these tests also indicate that there are some problems with the C implementation of the client stub of the dispatcher, as there are performance problems present when sending 32 KB of data through the dispatcher, which are absent in the Java version.

The main point in where the C/C++ implementation proves its value is in the areas where cryptography or data compression have to be done: the algorithms used for both platforms are the same for all of these encryption and compression tests, and Java simply can not compete with a C based implementation. Especially the MUX throughput, which is almost always limited by the encryption that needs to be done, is considerably faster in the C/C++ AOS.

However, whenever I/O latencies come up, the differences between both implementation become smaller. Even though the C implementation can encrypt more than 5 MB per second worth of data using 3DES, where the Java implementation cannot even reach 1 MB, all of this does not matter anymore whenever the inter-AOS connections are not capable of transfer at these speeds. Similar effects of I/O on performance could be seen in the AC finalize tests: if the C/C++ implementation keeps segments on disk, the performance leads become smaller.

Still, in its current implementation, there are no areas in which the Java AOS comes very close to the performance of the C kernel implementation.

As for the ability to have both implementations interoperate, this was at least shown by the fact that both implementation can understand each others requests in XDR, either over MUX/SSL or over the SunRPC dispatcher. The tests that were done between both implementations were not as extensive as the entire set of tests that were done for the C/C++ implementation, but all of the regular tests showed no unexpected results: both implementations worked together as they should do.

Unfortunately, at the time of writing, the C/C++ implementation is not entirely complete. Cookie persistence was not completely implemented due to a lack of time, but the rest of the functionality is all present. This does not mean that there are no optimizations possible: both the segment caching policy, and the base connection cleanup strategy are not optimal. These should preferably be optimized to take usage frequencies into account (LRU for segment caching), and base connections should really only be removed if other connections cannot be made otherwise. SSL session caching for base connections is also something that can be added for performance increase. The problem with low transfer speeds for sending 32 KB over the dispatcher should at least be traced, and preferably solved.

6 Future Work

The C/C++ implementation is complete in the sense that all the main functionality from the AOSv5 specification (except for crash persistence) is all provided. Implementing crash persistence might be one of the first additions that has to be made to the C/C++ implementation, but all the middleware systems that will be using AOS (e.g. Mansion middleware) must be able to handle this crash persistence.

There are no critical bugs in the implementation as far as the functionality is concerned, but a number of inefficiencies are mentioned in chapters 3, 4 and 5. The optimizations that can be made according to chapter 3 are:

- Better segment caching. A real policy (except for a first-come, first-serve) is absent, whereas a LRU scheme suit the needs for middleware much better. This should not be too hard, but the current decentralized caching structure might have to be revised to a centrally managed caching in order to implement an LRU scheme.
- Some optimization can be made to the AC finalize process. In the current implementation, a completely new AC ZIP file is created, even when many the segments are already in the ZIP file (from a previous finalize). Also persistent segments might directly be stored in the AC ZIP file for example. This will reduce the time needed for finalizing of an AC, especially if the agent that uses this AC takes multiple hops (and as such requires to do multiple finalize operations).
- Base connection management might need some better cleanup mechanism: instead of the fixed timeout before removing an unused base connection, it might be better to only remove a base connection whenever there is actually demand for this (e.g. out of file descriptors).
- SSL session caching could greatly reduce the handshake times needed otherwise for SLS connection setup. Currently this is not implemented, but it might certainly be a useful addition, as the SSL handshake has proved to be a large overhead in the entire connection process.

Chapter 4 revealed some unsatisfying performance numbers, and it might be worth identifying the cause of these problems. Also the tests were not entirely exhaustive, some more tests might need to be done to further analyze the performance of both AOS implementations. The important ones are:

- The performance of the SunRPC dispatcher (over TCP) seems to be quite poor in certain regions, especially when sending 32KB the performance seems to be extremely low compared to other sizes. This can be an issue in the libc implementation of SunRPC, but it is not quite clear by what this is caused. It might be very well worth finding out how this can be eliminated (maybe by using a different transport like Unix domain sockets).

- The virtual connection throughput tests, in which 10 senders send only a small amount of data, showed some bad throughput numbers. The total utilization of the channel for these tests is not good compared to the other tests (the tests with more data or fewer senders). Where this problem comes from is unknown, but investigating this (maybe using micro benchmarks in the AOS code) might help to solve this problem.
- All performance tests were done on a single machine. Tests using multiple machines, connected by various types of network can give a more realistic overview of the performance of virtual channels in real-life situations.
- Using a different operating system (and file system) could give more insight in some of the performance numbers, and might lead to different conclusions for various configuration. Some other things that can be tested are different operating systems (for example Solaris, on which even dispatcher tests with the automatic multithreaded mode can be done) or using a different (more traditional) file systems to see if this influences the finalize times.
- Finalize can be tested for agents that have different content: some middleware systems host agents that collect only text or images, or some other kind of data. Such tests could provide better performance numbers for different applications.

The testing section was only meant to highlight the most obvious performance characteristics of both implementations, but it was not in the least extensive. More micro-benchmarks could have been done, for both kernel implementations, as well as experimenting with different JVM implementations (with JIT enabled). More extensive tests are all left as future work, depending if there is a need to have a more thorough analysis of the AOS performance.

The AOS version 5 specification has at least one clear omission, and that is flow control for virtual connections. The current implementation uses a sub-optimal flow control based on TCP flow control, but it is only a workaround. Any new version of the MUX specification should preferably at least include a simple flow control mechanism, so that the buffer sizes can be managed for each individual MUX connection, instead of managing MUX flow control relative to each base connection. Because the MUX protocol itself is quite simple, this flow control should preferably be simple as well. A version 6 (or later) of the AOS specification could for example introduce a (fixed/dynamic) buffer for each MUX connection, and introduce some extra control messages using which buffer information can be exchanged.

Appendix A: required technologies

This appendix will introduce the various technologies that were used to realize an implementation according to the specification of AOS. Technologies such as the C++ programming language basics (both regular ANSI C and C++ classes) and network programming using TCP sockets are assumed to be known already.

Extra technologies that were needed for this implementation of the AOS kernel are: the use of external data representation (XDR) for platform independent interaction, SunRPC for inter process communication (IPC) between AOS and middleware systems running on top of AOS, OpenSSL for all encryption-related problems (creation of signatures and checksums, encryption of communication channels), the pthreads library for adding threading support in the AOS kernel, and the ZipArchive library for data compression.

A.1 External Data Representation (XDR)

When programming an application that utilizes networking, a programmer may come across machines that have different architectures and operating systems. The implication for these kind of applications is that different machines often have different representation for their native data types. For example, the byte ordering of an x86 class machine (little endian) is different from that of a Sun SPARC machine (big endian). This means that the x86 class machine cannot correctly interpret binary data sent from a SPARC class machine. Also, different operating systems and programming languages can have dissimilar ways of representing data types: an int type value is not guaranteed to be 32 bits long on all systems.

If all systems send out (and expect to receive) data in their own native format, it is almost impossible to guarantee that all machines will be able to cooperate. Data sent out in big endian format may seem like garbage to a little endian machine. Or worse: data can be completely misinterpreted resulting in undefined behavior. What is needed for these kind of applications is a mechanism that allows all participating hosts to correctly interpret data sent by the same program running on a different host. This problem is addressed by the External Data Representation (XDR) format [RFC 1832]. The XDR standard defines standard formatting for all kinds of data types, which are translated to and from the machines native representation whenever data is sent out or received from another node. For example, the xdr_int() function can translate between the native and external representation of an integer.

The beauty of this all is that this xdr_int() function is present in a standard library on most, if not all, different UNIX-like platforms. However, the actual behavior of this function is determined by the platform it is running on: on a certain machine this function might not do anything if the byte ordering and int size on that platform are already the same as specified by the XDR specification. On another machine, this xdr_int() might actually change the byte ordering. Any program sending out integers using xdr_int() will compile on different platforms and the data sent over the wire will be correctly interpreted by all receiving parties. This way it is easy to write programs that can interact even though they are running on different hardware platforms: use the XDR routines whenever data is sent or received and let these XDR routines determine whether some reordering needs to be done. This way, a program needs not to be rewritten in order to run on different platforms, although the design

of the program might need some modification to fit this model, even though the model is quite simple.

XDR is used in the AOS specification to define the message formats for the multiplexed protocols. This way, all different AOS kernel implementations can immediately interact will all implementations, even though these can be implemented in a different language and/or running on a different platform.

A.2 SunRPC

The concept of a remote procedure call (RPC) is in its very basic a mechanism to allow a running process to communicate with a remote host [RFC 1831]. This is done by a mechanism that looks a lot like a regular (local) procedure call from the user?s point of view. Instead of that the function is executed directly in the same process, the function call (including its arguments) is sent over the network to a process running somewhere else. This remote (server) process will execute the requested function itself and whenever it has the results ready it can send them back. After receiving the result, the host that requested the remote call can continue with its own work.

In the ideal case, the programmer using such a remote procedure call does not need to be explicitly aware of the fact that its request is actually sent over the network to be processed somewhere else. Other than a little longer waiting time than what is usual for a local call, the user will not notice anything. Of course it is not bad to have the user at least being aware of some distribution of work that is going on, but it is nice to have a mechanism that could provide this as transparently as possible.

SunRPC contains mechanisms to provide for this in some level. What is needed for this is a specification of the methods that are to be accessible remotely. This specification is written in a platform-independent specification: a program specification in SunRPC is actually written in XDR. This specification is then translated into client-side and a server-side parts by the rpcgen program, which can then be integrated into the application. For more complex data types than the simple primitive like int and char, the XDR specification can also contain structs and unions, for which XDR routines are also created.

A client can simply call the client-side functions that are generated by rpcgen, which will take care of the sending and receiving of the data. On the server side, the generated code takes care of this in a similar fashion, it just acts as a layer on top of the native server functions.

A.3 OpenSSL

The Secure Sockets Layer (SSL) [3] and Transport Layer Security (TLS) [RFC 2246] protocols provide a layer on top of regular TCP connections, extending these with encryption and authentication in addition to the networking features of TCP (flow control and lossless operation). The basics of cryptography and such will not be described here, but the OpenSSL library offers a wide variety of cryptographic protocols to ensure the secrecy and integrity of data channels by the use of encryption and message authentication. Also, peer authentication is offered by OpenSSL, both for the client and server sides of the connection (depending on whether they have a certificate). How all these technologies work is far beyond the scope of this thesis (as are a lot of other features that OpenSSL contains that

are not even mentioned here). For an extensive introduction to the world op cryptography refer to [4]. For a more practical introduction to SSL programming please see [5].

Anyone creating an SSL/TLS connection can decide how strong the encryption is by setting of a cipher suite prior to making a SSL/TLS handshake. A cipher suites is a combination of a key exchange algorithm, an encryption algorithm and a message authentication code (MAC) algorithm. But OpenSSL does more than only offering encrypted channels: it also has a rich set of mechanisms for more generic cryptographic operations: it offers hash algorithms, public key and symmetric key cryptography, it offers mechanisms to generate keys and certificates, and allows the creation and verification of signatures in various portable formats.

A.4 Pthreads

Usually, a running process can handle one task at a time. However, for networked programs it is not unusual to have a need to handle different operations at a time. One part of the program might want to wait for connections to come in over a TCP socket. Usually these kind of operations are blocking, which is very inefficient if you want to have a program to handle much work. Some kind of polling can be used, but this is not always the most efficient mode of operation.

If one needs to have multiple tasks running at the same time run, different programs can be run at a time, which can pass information over some kind of inter-process communication (IPC) channel or shared memory. However, if all of these actions are part of a tightly-coupled application, this may be a little hard (if not impossible) to accomplish using multiple processes, as each process runs in a separate address space and cannot have access to variables in the address space of another processes. Shared memory can be used to share data between processes but this may be hard to accomplish for large-scale systems. Moreover, having many separate processes running can also be very inefficient because and communication has to be done using IPC, which is clearly more costly than using regular function calls. Not to mention that every single communication channel can introduce possible security leaks. Also, the applications becomes fragmented an may become really hard to maintain.

Another approach is to have more than one thread of execution per process. These threads can be seen as process-in-a-process and rely on operating system support (the thread scheduler) to make sure there threads are scheduled according to a certain kind of policy. The main advantage of having multiple threads in a single process is that a multithreaded process can handle multiple tasks at the same time, while all of these threads have access to the same data: they work in the same address space. On uniprocessor machines, these threads are not actually all running at the same time, but the thread scheduler makes sure that only one thread at a time is active in a time-share fashion. One thread is running at a time and the rest of these are suspended. Once a thread has used up its time slice it will be suspended by the scheduler, after which another thread will be started. Threads that execute a blocking call can also be suspended in favor of threads that are not blocked.

Creating a multithreaded program has a lot of advantages. However, it also has a number of disadvantages, mostly because raises the level of complexity. Because threads are scheduled differently on each program run the process becomes non-deterministic, which makes it harder to debug as errors may be hard to reproduce. Moreover, protection of internal data is a serious issue. If one thread is busy with writing some data and halfway the scheduler

activates another thread that needs access to this same data (race conditions), consistency cannot be guaranteed.

Why threading support is so important for AOS is for a number of reasons. First, it manages many connections at the same time (both incoming RPC calls from the middleware level) as well as many SSL connections. Monitoring all these with a select call is a tedious task, and can only be done by means of some low-level programming, at least for incoming RPC requests: these should all be handled by hand. Second of all, AOS offers (and uses) a number of blocking calls. Mostly these have to do with IO operations. While it might have been possible to circumvent some of these problems, this is nearly (if not completely) impossible. Also, multithreaded programs can offer a better utilization of resources, as otherwise all calls have to be executed one at a time. Bottom line is that a realistic and fully functional implementation of AOS without threading support is not reasonably possible.

For AOS, threading support was added by the pthreads (POSIX threads) library, which have the advantage of being supported on many different platforms. A program using this kind of threading will run the same on all platforms that support pthreads (minus differences in scheduling). A new thread can be created by giving it a function to start the execution, and possibly some arguments for that function. This thread runs until either this function finishes, or the thread calls the pthread_exit function. During its lifetime, the thread can be scheduled and suspended many times, but all this scheduling is hidden from the programmer.

The only thing the programmer really needs to be aware of is the fact that each thread has its own stack and program counter. Arguments for functions are passed on the stack, so multiple threads running the same function will not interfere with each other (if the arguments do not refer to other memory areas). However, the data that is stored on the heap (for example, memory allocated by malloc or new) is shared between threads, as are global variables and fields in a function that are declared static (which are also located on the heap). These fields are vulnerable to inconsistencies due to concurrent access.

A.2.1 Mutual Exclusion using Mutexes

Fortunately, the pthreads library offers a variety of mechanisms to help with achieving mutual exclusion of access to data. The most important element to achieve this is a mutex, which can be seen as a lock on data. The mutex can be locked and unlocked by different threads, but only one thread can hold the lock at a time. If multiple threads at the same time attempt to obtain the lock, only one will get the lock and the other threads will remain blocked (suspended by the threads scheduler). Once the thread that has the lock releases it (unlock) one of the others will unblock. There also exists a non-blocking variant, which also can be useful sometimes. Note that mutexes do not guarantee that the data that requires the lock is not altered: if one thread accesses the data without having the lock it can do so, but at the risk of running into inconsistencies. When programming multithreaded programs with data sharing between threads, the programmer always needs to be aware of what data is protected by which mutex.

Also, once a mutex is no longer needed, it must be unlocked. This is not done automatically, not even after the owner thread exits. If unlocking is forgotten once, this will likely stall the entire process. Also, not all platforms support the multiple locking of mutexes: if a thread wants to lock a mutex it already owns, it is very likely that this operation will block forever. These are some of the difficulties that are introduced by threads programming.

A.2.2 Condition Variables

Another very useful feature offered by the pthreads library are condition variables. These offer yet another way of synchronization between threads, but not by only controlling access to this data: the access is controlled also by the value of the data. This is very useful for blocking function calls, for example if a thread is waiting for some data to become available. Such data is generally protected by a mutex, so a thread could repeatedly lock the mutex, see if the data is available and if it is not, release the mutex and sleep for a while before trying again. This is a resource-wasting strategy because the thread waiting for the data is being scheduled constantly without to poll the state of the connection without any data being available.

A condition variable is a device that can be waited on by a certain thread, and signaled whenever the condition is met, unblocking the thread that was waiting on that condition. This eliminates the polling strategy that should otherwise be used, which is a good thing because it also simplifies the model. Condition variables are relatively easy to understand and quite elegant in their use.

A.5 Zip data compression

Lossless data compression is useful for saving storage space as well as saving space for reduction of the network load. Although it requires some processing to compress the data, as well as some extra time to decompress the data, this extra cost is usually compensated by a reduction in IO times, especially when the data has to be sent over a network.

Zip compression is standard present on a variety of Unix and Linux platforms in the *zlib* library. Unfortunately, this library is a little difficult to use when storing multiple files in an archive file (conforming to the DOS PKZIP format). Luckily there are a variety of libraries present that offer a little more high level API to *zlib*. ZipArchive is such a library and it offers a remarkably simple (C++) interface. The basic functionality is quite easy to understand: a compressed archive (a .zip file) contains multiple compressed files. ZipArchive can do a lot, but only simplest mechanisms are used for AOS: storing to and extracting files from an archive.

ZipArchive is included as a part the AOS distribution, and it is available under a GPL license. Actually the ZipArchive distribution also supports a variety of platforms such as Windows based systems. Supplied in the AOS distribution is a subset of the library, namely only the Unix/Linux parts of the implementation.

ZipArchive is a product of Artpol Software, it can be found at http://www.artpol-software.com

Appendix B: Programming on AOS

This appendix will present some of the actual details and problems encountered while making the C/C++ implementation. One could think that using the AOS specification it is relatively easy to convert this to a running AOS kernel implementation. This is partly true. While the specification does contain some implications for the architecture, a lot is still left for the programmer to handle.

This appendix will start with a small section on the code organization, and where some compile-time limits can be set in AOS. The rest of this appendix will present the most important programming details of the AOS kernel implementation, and is as such probably only interesting for people who need to modify certain parts of the implementation itself, and so this is more of a “programmer’s guide” then an actual architectural motivation.

B.2 contains an introduction as to how the implementation was made so that all of its functions (and data) are thread-safe, which is the most important base for building a multithreaded application: threads should not interfere with each other. Operations on the same data sets should not be interleaved, but processed in-order. These mechanisms of protecting the data structures are quite generic, and are used in the same way throughout the AOS system.

Section B3 contains details on how the multithreaded SunRPC dispatcher was developed. Usually SunRPC only has a multithreaded extension when it is running on Solaris. Systems like Linux do not have this functionality, so the dispatcher had to be designed in a different way to ensure portability of this multithreaded dispatcher. A multithreaded dispatcher is essential to AOS for reasons explained in section 3.3: a single threaded solution is inefficient and blocking operations could cause the entire system to halt.

Section B4 contains details on how to ensure the use of correct cipher suites using OpenSSL.

B.1 AOS Code Organization

The AOS code is divided in a few parts. The most important parts concerning the AOS kernel itself can be found in the source release in the directory src/kernel. The directory src/xdr contains the XDR specification of the protocol, src/aosclnt contains a client-side stub which can be used for clients written in C to contact the AOS kernel, and src/libaos contains some generic functions that can be useful for clients as well as the AOS kernel.

Compile-time options for the kernel (such as internal limits) can be set in the AOS configuration header (src/kernel/include/aos/aosconfig.h). Most of these are quite self-explanatory, but comments are placed whenever necessary.

The implementations used to be C only, but halfway the development some of the data structures were rewritten as C++ containers, mostly for offering better re-use of mutex-protected resources and for offering automatic mutex unlocking for these protected data structures. For the implementation of the mutex-protected containers see appendix C.

B.2 AOS data structures & thread-safety

The protection of data from concurrent access is a key issue. We will no look at the way the correctness of AOS its internal data can be ensured by the use of mutexes to control the access to data. This will be done with the use of some generic examples about the concurrent access to role data, but the idea is the same for other critical data structures (e.g. segments, agent containers, connections etc.).

Basically, all of AOS its internal data structures are modeled as structs (or classes) containing the information for that part of the data. This is all common practice in C (or any other kind of programming language). What is added to these structures are a mutex for each one of these structures. Usually these structures are stored in some form of array, which also has a mutex associated with it. Whenever a thread wants to look up or manipulate this array, it must lock the mutex for this array first. Once that is done, it can get see which elements are present in the array. It can then lookup the memory location of the specific structure it needs. To actually *use* this structure itself, it also needs to lock the mutex of this structure.

Once a thread has a lock for the struct, it can perform its operations on the data. AOS contains many different types of data that are all protected from concurrent access in this way. For example, roles are all represented internally to AOS as structures with the role data. In addition to this role data, such a structure also contains a mutex. A simplified example is shown below:

```
struct role {
    int id;
    cookie c;
    pthread_mutex_t mutex;
}
```

Example B1: Role structure

```
role** roles;
int arraysize;
pthread_mutex_t mutex;
```

Example B2: Role table

So, suppose someone wants to insert a new role into the role table called roles. This requires a lock on the role table, because that action implies modifications to the table. After this is done, the new role struct can be inserted, and all of the mutexes can be released. The next example contains the source code outline for how to do this (minus error checking).

```
void insert (role* r) {
  pthread_mutex_lock (&mutex);          // lock role table
  segarray [arraysize++] = r;           // insert role
  pthread_mutex_unlock (&mutex);        // unlock role table
}
```

Example B3: Inserting a role into the table

This is a fairly trivial example, where we assume that the calling thread already has safe access to the role structure it wants to insert. Once the role is stored in the table, other threads will also be able to request access to this structure. Note that mutexes do not really prevent access to the table: if someone wanted to then the same code without the pthread_mutex functions would work equally well. Except when not running in a concurrent environment, it

is not guaranteed that another thread is not scheduled in between and also inserts something, probably resulting in data loss. Forcing all threads to use only this method when inserting a data prevents these kind of problems. What happens when someone wants to obtain a lock on a role itself (which is in the table) is shown in B4

```
role* get_role (int index) // incorrect version
{
  pthread_mutex_lock (&mutex);             // lock role table
  role* result = rolearray [index];        // get pointer to role
  pthread_mutex_lock (&result->mutex);     // lock struct (blocking)
  pthread_mutex_unlock (&mutex);           // unlock role table
  return result;
}

void do_something (int role_id) {
  role *r = get (role_id);            // obtain role pointer + lock
  do_work_on_role (r);
  pthread_mutex_unlock (&r->mutex);
}
```

Example B4: Locking a role from the table (incorrect)

Now the semantics of these kind of `get` functions are as follows: if and only if this function returns a pointer did a structure with such an ID exist. Not only did it exist, but the calling thread now *also* owns the lock on this structure. This also means that the function that receives the pointer should *always* unlock the mutex when it is done. If this is not done, the lock will remain intact, and any other thread waiting to lock the mutex will block forever. The semantics of such `get` functions are chosen as the standard convention for all AOS data structures.

Moreover, whenever a function (such as do_work_on_role) *receives* a pointer to such a protected data structure, it may assume that this pointer (and thus the lock) was obtained in a safe way. So, it can just assume it can access the data without the need for some other lock. To ensure consistency, it is best if the unlocking of the mutex is done in the *exact* same function as the one that obtained to lock. Other functions that are called further below need not to be bothered with this. It is even better if underlying functions do not unlock this mutex at all, because its caller might still need work on the data (except in some special cases).

It is easy to forget to unlock a mutex, especially if there are more ways to return from a function. Before each one of these exit points, the mutex has to be released. It is easy to make mistakes in this scheme, which will ultimately result in a total block of all parties that want access to the struct. One thread that forgets to unlock can stall the entire processing. Fortunately, it is also possible to develop a scheme that transparently handles the locking and unlocking of such data structures in a completely automatic fashion. The class for this is called lock_ptr and the idea behind this is quite similar to the standard C++ auto_ptr class. Such a lock_ptr can be attached to a pointer, and after the lock_ptr goes out of scope, the structure it references to is automatically unlocked as well. This is a recommended way of working with AOS its internal data structures. The description of the lock pointer and how it can be used is given in appendix C.

What is done in the get function seems simple, but now already we have a potential problem. This is quite a disappointment, as this example does not really do any complex operations yet (only data access). The problem is that even though this get function does not modify the role table itself, it *does* need to lock the role table in the first line of the get function. Otherwise, if some other thread modifies this table while we are trying to obtain a pointer to some role we might have a problem: we really need the lock for the table here also. After this we can safely try to get the lock for the role.

Simple as this might seem, a performance problem is already present in this example. If the role at position index is still locked by another thread, the current thread will *also* block while waiting for the lock to become free. However, it already holds another lock (the mutex for the role table), which means that other threads that want to get access to *any* other role will also have to wait until line 3 of the get function completes. If the process of waiting on the role mutex takes a while, this will ruin any kind of performance, and it does so for all threads.

Even worse: whenever the thread that is keeping the lock decides it wants to destroy the structure, we have an even worse problem: how can we destroy a mutex when other threads are also blocking for access to it? There is only one answer: this cannot be done, as the behavior of destroying a mutex under these conditions are different for all platforms. So, all in all, this example will not work: it has dreadful performance disadvantages and behaves differently on different platforms. Not exactly a good property of a portable implementation.

Fortunately, there is a non-blocking variant for locking a mutex: pthread_mutex_trylock which makes this kind of data access possible. What this function does is try to set a lock and as a return value it returns *immediately* whether it succeeded or not. If it did not, the calling thread should just try again later, after which it can at least let go of the mutex for the table. The outline for such a function is given below, and is conceptually equal to what is used in the C/C++ implementation.

```
role* get_role (int index)  // correct version
{
  for (;;)
  {
    pthread_mutex_lock (&mutex);

    // trylock returns 0 on success
    int lock = pthread_mutex_trylock (&rolearray[index]->mutex);

    pthread_mutex_unlock (&mutex);

    // the table mutex is now again free for other threads.
    // Only now can we check whether the locking actually succeeded

    if (lock == 0) break;         // done, success
    usleep (POLLING_INTERVAL);    // try again later
  }

  return rolearray[index];   // lock obtained, can return reference to caller
}
```

Example B5: Correctly locking a role from the table

This example is again a little simplified (it does not contain error checking, especially for deleted roles), but it demonstrates the need for the non-blocking locking variation. In the first line of the loop, the table is locked after which we try to quickly lock the structure itself. After this immediately the table is unlocked. Should we have succeeded in obtaining the lock, we can jump out of the loop. Otherwise, try again later. Note that with a simple modification this also works if the role we are waiting should be deleted in the meantime.

Note that some form of polling is used: it is the only place in the entire AOS implementation where this is used. And even though it could maybe be replaced with a (complex) implementation using condition variables, this one is simple enough and works well enough. Moreover, its behavior is correct on all platforms and if this can be simply done by polling the mutex once in a while, this is fine.

The actual implementation is a little more extensive, but roughly follows along the same lines. Because these locking strategies are similar for most cases, there are a standard set of containers that can be used for storage of data that needs to protected from concurrent access. See appendix C for the listings of the protected data structures. In the C/C++ AOS implementation, classes contain the arrays, and access to the elements is automatically regulated. All elements are also classes instead of structures. This allows for better code reuse of this locking rules. The class SynchronizedTable is the base class for all tables that are protected by a mutex (you can define any kind of table, like a direct-indexed table for example). Such a synchronized table stores data of type SynchronizedElement. A synchronized element can be anything like a segment, agent container, role etc. The class constructor can be optionally given an argument to not use the mutex, which may be useful for using protected structures within other protected structures, which is the case with an agent container (it contains a list of segments). For all these data structures see appendix C.

However, the use of these containers was added relatively late in the development process (the implementation did not use C++ for a long time). Some traces of the old approach can still be found, which was basically the same but without the use of classes to contain this functionality. Using classes allows for better code reuse because it offers inheritance and a higher level of abstraction. At the time of writing, not all parts of the data structures are rewritten using classes. This is not always that simple, as it sometimes requires relocation of code which is not always trivial.

B.2.1 Concurrent Data Access with Blocking Function Calls

There are a number of blocking calls in AOS, which use condition variables to wait for some condition to occur. These functions are accept, read, peek, wait_ac and select. Whenever a client calls a function to wait on such a condition, the mutex is automatically unlocked and another thread can access this structure. A possible problem here is that it is not really desired to have *multiple* threads perform a blocking call on the same data (like multiple blocking accept calls pending on a single endpoint descriptor). Especially when one thread has the intention to delete a data structure and some other thread is still waiting for it. To prevent these kind of circumstances AOS limits all of these blocking calls to one per data structure. It is of course possible to have multiple accepts pending for different listen endpoints.

The only exception to this rule is the select call. Even though select is limited in the AOS specification to one pending call per role, this limit is a little higher in the C/C++ implementation. A role can in fact have more than one select pending (though still a limited amount) as long as the sets of descriptors that are to be monitored do not overlap. There are two ways of calling select, blocking or nonblocking, and this limitation is only present for the blocking variation. Without blocking, this is not really a problem. However, because of the way connection monitoring is implemented, a role *can* have a blocking accept call pending on an endpoint, as well as having a select blocking on that endpoint. Whether a middleware process will actually do a simultaneous select and accept call remains to be seen, it does not seem like an entirely realistic scenario.

Whenever a role wants to call a blocking function on some data, when there is already some other thread blocking on the same data, AOS returns an access denied error, except for calls to wait_ac, which has dedicated errors for this.

B.3 Dispatcher Implementation

The rpcgen program was used to generate the AOS server-side stub for SunRPC. However, a standard dispatcher generated by rpcgen is not always good enough, at least not for multithreaded applications: a server skeleton generated by the standard rpcgen can handle only one request at a time. This is at least inefficient, but also unusable if the dispatcher offers access to blocking functions. There are some blocking functions in the AOS API so a single threaded dispatcher is not usable for this purpose: one blocking call could prevent all access to AOS during the time that the function is blocked. If the function blocks long (or, say, forever) we have a major problem.

Although Solaris offers a multithreaded extension, this is not standard and thus cannot be used on platforms that do no have this extension. For example, Linux does not contain this

functionality. The difference does not only lie in what is generated by the rpcgen program on these platforms, but the RPC libraries on these platforms have different functionality. So, unfortunately, the Solaris extensions are no solution for all other platforms.

Fortunately, there is a solution that does work for all platforms with pthreads support. It is only a little harder to use: threads must be manually managed for each request. Because rpcgen generates server and client as C source code files, these can be manipulated by hand. Whenever a request comes in, the skeleton interprets the arguments and decides what function to run. After this is done, the actual function is executed and then the results can be sent back.

It might seem quite trivial to insert the creation of a new thread somewhere in between there. In practice this is a little harder. Especially because the data structures that are used for handling incoming RPC function calls are not thread-safe, meaning that these will be overwritten whenever a new request comes in. It is not always very well documented what all parts of these data structures do. To illustrate, whenever a request comes in, the following function is called for the AOS SunRPC dispatcher:

```
static void aos_api_5 (struct svc_req *rqstp, register SVCXPRT *transp);
```

The first argument contains the function information (most importantly a function identifier), and the second argument contains transport information of the peer that requested the function (socket and connection type). The pointers to these structures are probably pointers to local data on the stack of the calling function in the RPC library, which means these cannot be shared between threads. So, whenever a new RPC request comes in, these values are overwritten. This means that we can either delay the passing on of the data to a worker thread until these are not needed anymore, or we can copy them, after which the structures are at least safe from being overwritten. Delaying the worker thread is not a possibility, because these arguments are also needed to send back the data. So, this can only be done once the function call finishes, which would effectively degrade the dispatcher to a single threaded one. So copying the arguments for the new thread seems like the way to go.

However, this is not enough. The library containing the SunRPC code (libc on Linux systems) reuses socket descriptors whenever this is possible (actually, the client notices that whenever it receives an xdr_endofrecord notifier it can reuse the connection). If a new connection is used, registering of this new connection is not thread-safe at the server side, so some of the code from libc was copied and moved to AOS internally.

This code inserts locks on file descriptors, to make sure each TCP connection can only be used by one thread at a time. Whenever a requests arrives at the custom svc_run routine (see mysvc_run.cpp) a lock is set for the file descriptor (see fdlock.h). However, even if the poll function in mysvc_run sees data arriving on more than one socket, only one file descriptor is activated at a time, and so the locking is done in two phases: mysvc_run sets a *transient* lock, which can be reset multiple times, but only when the request arrives in the dispatcher this lock is made *persistent*. This persistent lock means that other locks (also transient locks) can only be set after the persistent lock is released. This way the file descriptors are correctly synchronized.

This kind of programming work looks a little tricky because it relies on some knowledge that is not really documented in the RPC manual pages: it requires analysis of header files and

library implementations to find out what exactly is done where and doing locking at just the right place. The largest advantage of this approach is that (when it works) it works on all platforms independent of whether the RPC library has multithreaded support or not. Pthreads are all that is needed for this apart from a single threaded RPC library, so it really is a generic solution for more than just the Solaris platform. Portable or not, it is admittedly a little bit tricky. Threading support however, is essential for AOS, so customizing the RPC dispatcher saves us from using some other custom written dispatcher.

Should the AOS API specification change, then running rpcgen over this renewed specification will not automatically create the correct server-side code. Although most versions of rpcgen generate quite similar code, there are some options that can be changed. For this implementation, the Solaris version of rpcgen was run with the -MA option (which generates thread-safe code). After this, all the changes mentioned above were applied. This -MA switch is currently not present on systems like Linux, so when changing the dispatcher (API) specification, a new skeleton has to be generated, after which the changes are best merged with the existing API by hand.

B.4 OpenSSL Cipher Suite Negotiation

The OpenSSL package provides the SSLv3 and TLSv1 functionality. There is however a little difference in how cipher suites are specified in the SSL and TLS RFC documents. See for the differences between RFC and OpenSSL formats see the following table:

RFC format name	OpenSSL format name
TLS_RSA_WITH_NULL_MD5	NULL-MD5
TLS_RSA_WITH_NULL_SHA	NULL-SHA
TLS_RSA_WITH_RC4_128_MD5	RC4-MD5
TLS_RSA_WITH_RC4_128_SHA	RC4-SHA
TLS RSA_WITH_IDEA_CBC_SHA	IDEA-CBC-SHA
TLS_RSA_WITH_DES_CBC_SHA	DES-CBC-SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA	DES-CBC3-SHA
TLS_RSA_WITH_AES_128_CBC_SHA	AES128-SHA
TLS_RSA_WITH_AES_256_CBC_SHA	AES256-SHA
SSL_RSA_WITH_NULL_MD5	NULL-MD5
SSL_RSA_WITH_NULL_SHA	NULL-SHA
SSL_RSA_WITH_RC4_128_MD5	RC4-MD5
SSL_RSA_WITH_RC4_128_SHA	RC4-SHA
SSL RSA_WITH_IDEA_CBC_SHA	IDEA-CBC-SHA
SSL_RSA_WITH_DES_CBC_SHA	DES-CBC-SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA	DES-CBC3-SHA

Table B6: Differences between RFC and OpenSSL cipher names

This table contains all the ciphers that are supported by the C/C++ implementation of AOS. What such a cipher string actually contains is a combination of the connection method (SSL or TLS), the key exchange algorithm (always RSA in this case), the encryption algorithm and the message authentication (hashing) algorithm. For example, the cipher string TLS_RSA_WITH_RC4_128_MD5 indicates the TLS protocol with an RSA key exchange, and RC4 with a MD5 message digest. However, these names are represented a little different internally to OpenSSL, as the right column shows.

The AOS specification uses the name formatting as used in the RFC documents. Now this need not be a problem if there is a one-to-one mapping between these different representations, but obviously this is not the case. OpenSSL ignores the prefix of the connection *method* (SSL or TLS) and instead leaves that to be defined when defining the actual allowed connection method. So, when a connection was is with OpenSSL using the RC4-MD5 cipher over a TLS connection this results in using the cipher TLS_RSA_WITH_RC4_128_MD5, or at least so it is reported to the client. So, AOS can force the connection to use TLS only if this is desired by setting the method of the connection to TLS only. This seems all very simple but we are not there yet.

An application can specify multiple cipher suites it finds acceptable for a connection. It can also decide to allow SSL for one set of ciphers, and TLS for another set of ciphers. The picture changes a little in this case (figure B7):

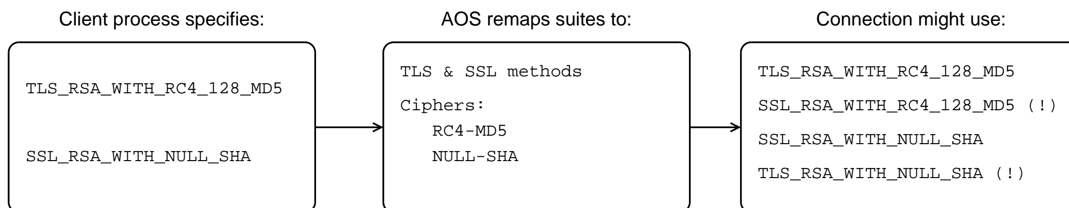

Figure B7: Problems with remapping cipher name formats

Why this is a problem is because when setting up a connection using OpenSSL, one needs to define all allowed methods for this connection *and* all allowed ciphers for this connection, but as separate features for this connection. After this, both parties can negotiate what ciphers will be used. If OpenSSL is allowed to use both the TLS and SSL methods, it is allowed to use these methods for *all* cipher suites that were provided, but this can result in something different from what the user actually wanted. One could argue whether this is really a realistic case, but every user can have its own valid reasons for specifying certain kinds of cipher suites. It is strange, if not wrong, to have a different method for a connection than was requested, while having AOS still report that everything went fine.

Fortunately, there is also a relatively simple solution for this, though it is not the most elegant one. AOS can first choose to setup a connection with all SSL based ciphers it got from the client. If it succeeds in making a connection then all is well. Otherwise it can try to setup a connection again, but this time only with the TLS based ciphers. Once again, if it succeeds all is fine, otherwise both AOS instances cannot agree on a cipher suite to use. This is exactly the desired behavior, although this might require one extra (failed) handshake than we would desire. This is the price to be paid for guaranteeing that *exactly* the right cipher is used.

Of course, either SSL or TLS support could have been removed as well leaving a much simpler scheme, but this was not done both for cooperation with the Java kernel implementation (it used to support mainly SSL ciphers) and for the ability to use the AES ciphers, which are present only over TLS so it seems. Also, this property was discovered relatively late in the development process when most of the functionality for handling both SSL and TLS version ciphers was already present. Should these problematic cases occur too often in practice then SSL support can always be removed later.

Appendix C: Source Code Listings

Here are some sample source code listings that might help in understanding the standard containers which are used to ensure concurrency. This is useful if you want to find out how the implementation of the mutex-protected containers and elements works, but it is as such only useful (if at all) to someone wanting to do some programming on AOS.

All elements that are to be accessed by multiple threads should derive from the class SynchronizedElement after which they can be inserted into, obtained or deleted from any instance of a SynchronizedTable. The SynchronizedElement itself is fairly simple:

```
class SynchronizedElement {
protected:
   int id;
   pthread_mutex_t mutex;
   bool usemutex;

public:
   SynchronizedElement (bool usemutex);
   virtual ~SynchronizedElement ();

   virtual int getid () { return id; }
   virtual void setid (int id) { this->id = id; }

   virtual bool equals (SynchronizedElement *se);

   void lock ();       // blocks until lock is obtained
   void unlock ();
   bool trylock ();    // return true if lock obtained, and false otherwise
};
```

Listing C1: Synchronized element class

A class that is to be inserted into a synchronized table must derive from a synchronized element, though it might not use the actual mutex itself. Each synchronized element can decide itself if it wants to use the mutexes for locking or not, determined by the usemutex argument of the constructor. This is useful to avoid that there are multiple mutexes (over) protecting the access to the elements. Such is the case with for example agent containers, which contain lists of segments, which do not need to have a mutex themselves. The locking functions do not do anything in these cases. The synchronized table itself is quite simple:

```
class SynchronizedTable {
public:
   SynchronizedTable (bool usemutex);
   int insert (SynchronizedElement *se);
   SynchronizedElement* get (int id);
   void delete (int id);
};
```

Listing C2: Generic interface to mutex-protected table

The synchronized table also has a really simple interface, one inserts an element and if everything went according to plan, the insert routine returns a positive element identifier, which can be used later to retract the element from the table again. Should you want to delete an element from the table, then the delete operation removes the reference from the table again. Note that this does not destroy the referenced object itself: this has to be done manually because that can involve different operations for different kinds of elements.

There are currently two subclasses of synchronized tables implemented: a direct-indexed table (element ids are direct indexed) and a compact (unsorted) version, which searches in linear time for the elements. For some element types direct indexed is preferable, for others the compact version is more suitable.

When one wants to obtain a lock and reference for a certain synchronized element (e.g. an AC) then such a function could look something like in listing C3.

```
int finalize_ac (int acid)
{
   AgentContainer *ac = get_ac (acid);
   if (ac->finalized ()) // ac already finalized
      return AOS_ERROR_NO_ERROR;

   int err = ac->finalize ();
   ac->unlock ();
   return err;
}
```

Listing C3: Forgetting to unlock

If the agent container was already finalized, then jumping out of the loop before the end causes the mutex to remain locked. Unlocking mutexes is something that can easily be forgotten. This can be prevented by means of what is called a lock pointer in the AOS implementation. Conceptually this lock_ptr is almost identical to the auto_ptr class which is a standard part of C++. An auto_ptr is a class wrapping a pointer, which is automatically freed once the auto_ptr is destructed, for instance when it goes out of scope. Such a lock_ptr can ensure that the reference is automatically unlocked once the object is dereferences, for example on function termination. The outline for this simple class is shown below:

```
template <class T> class lock_ptr {
private:
   SynchronizedElement *se;
public:
   lock_ptr (T *se=NULL) { this->se = se; }     // constructor takes locked element

   ~lock_ptr () { unlock (); }     // destructor automatically unlocks + releases pointer

   T* release ();    // release reference, this does NOT unlock the element, not even
                     // when it goes out of scope (lets the user take care of this)

   T* get ();     // get reference

   void unlock ();     // unlock the structure manually + release
};
```

Listing C4: Template for lock_ptr class

The lock pointer is extremely simple, but also effective in its use. Using these classes for all access to protected elements removes the need to explicitly unlock elements, which ensures that it can no be forgotten. The listing of finalize_ac looks somewhat different when using the lock_ptr class, see below:

```
int finalize_ac (int acid)
{
   lock_ptr <AgentContainer> lp (get_ac (acid));
   AgentContainer *ac = lp.get ();

   if (ac->finalized ()) // ac already finalized
      return AOS_ERROR_NO_ERROR;

   return ac->finalize ();
}
```

Listing C5: Access to agent containers by means of a lock_ptr

So, using the lock pointer, it does not matter anymore how the finalize function terminates. Either way, the instance of the lock_ptr is destructed at the end of the execution of the function, which means that it releases the lock because that is what the implementation of the class destructor does. Once this function terminates other threads can access this element again. Should some nonstandard behavior be required (like element deletion) one can detach the pointer from the lock_ptr object by calling release() and the automatic unlocking will now not be done anymore.

It might seem that this class is not needed, because one can also always explicitly unlock an element. It also needs an extra line to obtains the pointer from the lock_ptr class. While this is true it makes sure that one can never forget this anymore. Note that this does not necessarily prevents all deadlocks, because if two threads are waiting for each other's locks can still block forever.

References

[1] Guido van 't Noordende, *Agent Operating System (AOS) API, version 5 rc 3*, Jan 11, 2005

[2] Alfonso Fuggetta, Gian Pietro Picco, Giovanni Vigna, *Understanding Code Mobility*, 1998, IEEE Transactions on Software Engineering

[3] SSL version 3 draft Specification, A. Frier, P. Karlton, and P. Kocher, "The SSL 3.0 Protocol", Netscape Communications Corp., Nov 18, 1996.

[4] C. Kaufman, R. Perlman amd M. Speciner, *Network Security*, 2nd edition, Prentice Hall, 2002, ISBN 0-13-046019-2

[5] Eric Rescorla, *SSL and TLS: Designing and Building Secure Systems*, Copyright 2001 Addison Wesley Professional, ISBN: 0-201-61598-3

[6] White, J. *Mobile Agents*. J. M. Bradshaw (Ed.), MIT Press, 1997

[7] Robert S. Gray, David Kotz, George Cybenko, Daniela Rus, *D'Agents: Security in a multiple-language, mobile-agent system*, Mobile Agents and Security, Lecture Notes in Computer Science, No. 1419, pages 154-187, Springer-Verlag, 1998

[8] Holger Peine, Torsten Stolpmann, *The Architecture of the Ara Platform for Mobile Agents*,In: Rothermel K., Popescu-Zeletin R. (Eds.), Mobile Agents, Proc. of MA'97, Springer Verlag, Berlin, April 7-8, LNCS 1219, pp 50-61

[9] G. Karjoth, D. B. Lange, M. Oshima, *A Security Model for Aglets*, IEEE Internet Computing, Vol. 1, No. 4, 1997.

[10] Anand R. Tripathi et al., *"Design of the Ajanta System for Mobile Agent Programming"* , Journal of Systems and Software, May 2002.

[11] F. Bellifemine, A. Poggi, and G. Rimassa. “JADE - A FIPA-compliant Agent Framework”. In Proceedings of Practical Application of Intel l igent Agents and MultiAgents (PAAM '99), pages 97--108, London, UK, April 1999.

[12] Guido J. van 't Noordende, Frances M.T. Brazier, Andrew S. Tanenbaum. *Security in a Mobile Agent System*, First IEEE Symposium on Multi-Agent Security and Survivability, Philadelphia, August 2004.

[13] Benno J. Overeinder, Frances M.T. Brazier. *Scalable Middleware Environment for Agent-Based Internet Applications*, Proceedings of the Workshop on State-of-the-Art in Scientific Computing (PARA'04).

[14] H. Nwana, *Software Agents: An Overview*, Knowledge Engineering Review, Vol. 11, No 3, pp.1-40, Sept 1996.

[15] Walter Binder and Volker Roth. *Secure mobile agent systems using Java: Where are we heading?* In Seventeenth ACM Symposium on Applied Computing (SAC-2002), Madrid, Spain, March 2002