Agent Operating System (AOS) API, version 5 rc 3.

Notes on the AOS Kernel Design and the AOS API
Status: draft

Guido van ’t Noordende, Jan 11, 2005.

Introduction

These design notes describe the architectural design of the Agent Operating System (AOS). AOS is a middleware layer that can be used to build higher-level (mobile) agent middleware systems (such as AgentScape and Mansion) upon. AOS manages agent containers (ACs), agent container migration and secure inter-middleware communication. AOS is intended to be portable, and supports different RPC dispatchers to allow different client (middleware) stubs written in different languages to access AOS.

This document describes the design, functionality, and (proposed) implementation of AOS. The full API is included and discussed in some detail. In some cases, AOS is discussed from the Mansion point of view, which is mostly noticeable by the use of C-style or SunRPC specification of interfaces including the AOS interface. Other systems (e.g., AgentScape) may describe the same interface using Java, or even XML-RPC style calls.

On this document

The current document is intended to give the full specification and philosophy behind AOS, but some changes may still be made due to discussions arising from this text.

Scattered throughout the document are several notes on implementation issues or potential improvements. Look for section headings starting with "On ..." Footnotes serve a similar purpose by expanding on (often implementation) issues aside from the main text.

1. Status of this document

The current (version 5[1]) AOS interface is based on discussions over the period May-October 2004 between the author and the AOS development team (i.e., Benno Overeinder, Patrick Verkaik, Arno Bakker, David Mobach, Olivier Marin, and Reza Haydarlou), to whom thanks are due.

The version **AOS API v5** is to indicate that it evolved from AOS API v3, written in September 2003, and v4, proposed in May 2004. Changes are discussed / noted in the text. There have been multiple different draft versions of v4 floating around, so we now skip to v5 to avoid confusion. The current document is based on the v4 draft document written by me in May 2004, and many discussions since, particularly with Patrick, Benno, Arno, and Andy. An important difference between AOS v4 and v5 is that, since v5, process management has been removed from AOS[2].

AOS API v1 and v2 have never been made public. AOS API v3 has partly been implemented but not in full. It is the intention that from now on all protocols will be published and have version numbers (i.e., starting with version 5) as part of the specification.

1.1 Acknowledgements

Thanks are due in various degree to Benno Overeinder, Patrick Verkaik, Arno Bakker, David Mobach, Olivier Marin, Reza Haydarlou, and Reinier Timmer for several contributions. Furthermore, thanks to Etienne Posthumus for many comments on earlier versions. Etienne wrote the first prototype of AOS in Python. Furthermore, thanks to Frances Brazier and Andy Tanenbaum.

1.2 Changes compared to v5rc2

AOSAPIv5rc3 is intended for internal use, mainly for programmer consumption.

Changes are noted using a **boldface marker** (e.g., **[new]** in the text.

1. I refer to version 5 in this document. This may be changed to a 3-digit version number like 0.0.5rcX in the near future. Version numbering is still under debate. Draft versions of this document which are still under internal review are postfixed with an rc ('release candidate') number, which will be dropped when the document is made public.

2. Note that AOS is sort of a misnomer now that process management has been removed from it.

2. Agent Operating System (AOS)

Agent Operating System (AOS) is designed to be part of multilayered mobile agent systems. AOS is a middleware layer which sits between the operating system (OS) and the application-specific middleware layer (AppMW) above it. AppMW provides an interface to agents using the application that AppMW is written for. AOS is intended to be a minimal (small) 'kernel' that provides basic functionality needed by most mobile agent systems to the AppMW layer.

AOS is generally expected to be implemented as a user-level 'kernel' process that runs as a separate (user-level) process from AppMW on every machine. This is shown in fig. 1. What this figure does not show is that some functionality (particularly, process startup and local file system access) provided by the local OS is used directly by the AppMW layer, bypassing AOS.

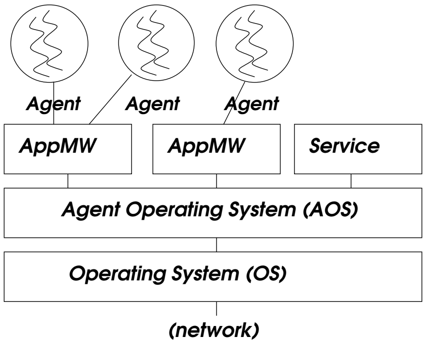


Figure 1. Middleware Layering

As AOS is really a middleware layer, it could conceivably also be compiled into the AppMW (e.g., for efficiency reasons). However, for protection reasons, AOS is implemented as a piece of software running in its own address space. This also allows for usage of one AOS process by different unrelated AppMW processes at the same time.

AOS can be implemented in any language. An RPC protocol over an IPC channel between AppMW and AOS is used to make invocations on the AOS API; therefore, AppMW may be written in a completely different programming language than AOS (provided that both programs use the same RPC protocol).

AOS may export multiple RPC protocol contact points and implementations to be used by different middleware systems (at the same time) if required. Multiple AOSes can run on a single machine, which each AOS serving one or more different middleware systems. Each middleware (e.g., Mansion Middleware (MW) an AgentScape MW) can consist of multiple different processes. Multiple independent (non-root) users may run different AOSes as user-programs on the same machine at the same time.

AOS provides services that we believe are required by most if not all higher level middleware systems that support mobile agents. AOS provices connection-oriented, authenticated, secure communication between different (middleware) processes on different machines, and an agent container construct in which an agent's code and data can be stored and shipped. The AC is also used to store associated information about the agent such as signatures of the agent's code and audit trails of changes to the agent.

First, we give a short overview of AOS and its relation to the operating system below it and application-specific middleware above it. Next we describe some architectural design issues and decisions for AOS. Finally, we give a per-method description of the proposed AOS version 5 API.

2.1 Design Rationale and Tradeoffs

AOS was designed starting from the recognition that Mansion and AgentScape have many similarities (as do most mobile agent systems). AOS implements a number of common core concepts on which more specific middleware systems like Mansion and AgentScape can be built.

2.1.1 AOS Requirements

- AOS should be able to interoperate with AOSes written in other languages or runnning on different systems. AOS should be reasonably efficient though only moderately so (it can always be optimized later), and it should support different AppMWs written in different languages. AOS always provides the same interface to higher layers, irrespective of the platform on which it runs. In that sense, it should provide portability to higher layers as far as AOS interactions are concerned.
- For interoperability reasons and portability, the API is kept simple and minimal. The interface uses no complex data types (i.e., no objects) and does not depend on platform specific abstractions like a specific file system interface. Furthermore, the protocols that the AOS layer uses to implement its functionality are defined such that different AOS implementations on different platforms can interoperate without any difficulty (those protocols will be described elsewhere).
- AOS is kept really minimal and provides only 'local' services that higher layers can use. AOS is not replicated over multiple machines, instead, each machine that runs AppMW components should run at least one instance of AOS. Through AOS, other AppMWs running on different machines, and (higher-level) services such as object servers, location services or other services that make an agent system work, can be accessed (connected to).
- AOS provides a mechanism for setting up a (secure) reliable, ordered communication channel to another AppMW process, possibly running on top of another AOS on another machine. The content of the messages exchanged between AppMW processes using such a communication channel are not interpreted by AOS, AOS simply provides a mechanism to securely route a (TCP like) byte stream to an AppMW. Different AppMW systems may use completely different protocols for exchanging information at the AppMW level using AOS as the transport level. Agent containers, in which agents and their associated data are stored, are also shipped over a, similarly set up, secure communication channel to an AppMW running on another AOS.

2.1.2 Design Decisions

It was decided that AOS should be minimal; AOS provides only services related to agent containers (ACs), AC migration, and inter-middleware communication. Process exection and agent execution is not made part of AOS as a single process or agent execution model was found to be too hard to get agreement on. Process management is now residing in the middleware layer.

Minimality also implies simplicity in the case of AOS. AOS is intended to be minimal and simple enough to allow for efficient implementations. Where possible, AOS maintains no state between operations.

An exception is that connection-oriented, ordered secure communication channels are provided to higher layers (on top of which those layers can implement an interface for semi-connectionless (datagram) like communication if they wish). This design choice requires AOS to maintain connection state. An important reason is that in Mansion, agents can set up reliable, ordered connections to each other. Secondly, internal to AOS, TLS (SSL) is used to provide secure communication.[3] Efficient security (i.e., using session keys) is easiest to achieve using connection-oriented communication.

Secure session state (of the internal SSL connections) is also stored by AOS although only for a limited period (session keys can always be recreated, caching of session information is a performance optimization but not a necessity. The system does not break if the session cache is cleaned up every once in a while (see Rescorla: SSL and TLS, Addison Wesley, 2001). This shows, as in some other aspects, that AOS is biased toward security in its design choices.

Another example is the decision to run AOS as a separate process from the application middleware. Although this is not strictly necessary (both layers can be compiled into a single binary program) we regard security as more important than performance in a mobile agent system. Moreover, latencies due to the distributed nature of the application is bound to dominate the performance, so that the extra overhead incurred by interprocess communication (IPC) is considered to be neglegible. This may be optimized for performance later. Optimizing for (or retrofitting) security in retrospect is a much harder task.

2.2 AOS Level Security

AOS provides a secure container for storing an agent and its data, called an *Agent Container (AC)*. The AC is used for shipping an agent between AOSes running on different machines. The AC is a small, portable file system, associated with an agent and managed by AOS, in which an agent's code and data is stored. An AC can be shipped to any platform running AOS to start the agent and access its associated data there.

In addition, AOS has mechanisms for setting up secure channels between AOSes over which AppMWs can communicate and ACs can be shipped. In addition, AppMWs need to identify themselves (using a credential, cookie) when invoking methods on AOS, which ensures that AppMWs can only access AOS resources (like ACs and communication channels) that they own.

2.2.1 Agent Container Design

The AC consists of a set of typed segments, essentially binary files. Every AOS has a table of

3. This is an internal implementation decision and may be changed later without influencing systems that use AOS.

contents (TOC), describing each segment. In addition, each segment is *checksummed* when an AC is prepared for migration (finalized), so that the AC's integrity can be checked. Each segment has a type. Example segment-types are *code* and *data* to store the agent's code in, resp. data that it saved.

Not all segments are used in every layer. For example, the code segment may be used by AOS or AppMW, but cannot be inspected by the agent. Agents can generally only inspect the data segments. AOS and AppMW can access all segments, in general (although, potentially, AOS could hide certain segments intended for AOS-level usage from AppMW. I don't see the need for this to happen, though).

Segments also have a subtype description, which can for example describe the specific code type in the segment, e.g., "Java_1.3_bytecode". A description field also exist which is usually used for naming a segment, which may be used also to further differentiate between segments of a given (sub)type.

Segments may be persistent or transient. In addition, segments can be encrypted. Transient segments may be changed or deleted. Persistent segments may not be modified. A transient segment can be made persistent but not the other way around. Encryption is expected to be used, in particular, to encrypt agent data segments for the agent's owner (those segments are generally also made persistent to avoid removal of the segment). Other segments may be encrypted also, though. Hop-to-hop encryption (of ACs on the wire) is already taken care of by means of the secure agent transport channel over which the agent is shipped to another AOS.

Each AC has a table of content (TOC) with an entry per segment containing its name, type, subtype, a persistence bit, and a checksum (SHA-1 hash) of the segment's content. Checksums need to be generated only prior to agent migration (using finalization, see above). AOS contains a basic AC integrity verification mechanism based on the TOC as part of the agent migration protocol with which an agent is shipped to another AOS kernel. Prior to sending an agent, the sending AOS finalizes and signs the AC's TOC using its private key. The receiving AOS can now verify the AC's integrity by verification of the checksums and the signature. If the AC checks out, the TOC is signed and the signature and the signer's certificate is sent back as a receipt by the receiving AOS. This finalizes the migration.[4]

Signing is mainly intended to facilitate AOS-level auditing; in principle, shipment of the AC over an encrypted, authenticated channel as is done in AOS already provides a degree of protection against tampering and privacy violation (on the wire). The signed receipt can optionally be stored by AOS (e.g., in a log file[5]), along with the TOC and the target AOS'es public key certificate with which the receipt was signed. These stored receipts can be used as a potential defense against claims that it sent a different AC than it did.

4. This is now included in the ACTP (agent container tranfer protocol), see appendix A).

5. Note that this log-file may grow to a considerable length. It is a local policy of the AOS owner / administrator whether and how long receipts are stored.

2.2.2 Self-certifying Identifiers

Principals are represented by public keys. Principals can be users, agents or processes. A *Self-certifying Identifier (ScID)*is a 160 bit SHA-1 hash of a (PKCS#1/RFC3280 encoded) public RSA key. The ASCII representation of a scid is the base32 encoded 160 bits SHA-1 hash of a (PKCS#1/RFC3280 encoded) public RSA key. The base32 encoding yields a 32 byte character string which is filename safe, i.e., the characters used in the base32 encoding contains characters that can be used in a filename or in a URL. The base32 encoding is defined in RFC 3548. ScIDs have a fixed size independent of the public key's length. **[new]** In AOS (e.g., in the endp struct), the unencoded binary representation of a ScID is used, consisting of 20 unsigned bytes.

The public key corresponding to a ScID is stored in a self-signed (x509) certificate. In the context of AOS, ScIDs are only used as 'endpoint identifiers' (EndpointIDs), which means that they are used to securely name AOS endpoints - single processes that have their own public/private key pair and a self-signed certificate containing the public key.

An other principal that can be named using a ScID (outside AOS) is an agent's owner. OwnerID is a shorthand for an agent or other entity owner - this is the SHA-1 hash of the public key of the owner. OwnerIDs can be used in access control policies or lists. An agent owner's certificate and OwnerID are sent along with an agent in its AC as part of the agent passport (see van 't Noordende et al., Security in a Mobile Agent System, 2004).

An EndpointID indicates an individual (nonreplicated) AOS or AppMW process or service, which has its own public / private keypair and a valid self-signed certificate. An endpoint certificate contains the process'es public key and has an expiration date. **[new]** The expiration date is determined by the endpoint owner when he or she generates the endpoint certificate. The validity of the certificate is checked at connection time.

EndpointIDs can be created and used in a completely decentralized manner. Anyone can create a new endpoint key-pair and certificate and publish the corresponding ScID (endpointID) in any way that suits it. In the case of AOS, however, the ScID is known to the AppMW process that uses AOS[6] and published to external parties (e.g., location services) as part of a datastructure (a `struct endp`). In addition to the endp struct, AppMW may also publish its own ScID (which may be a ScID for replicated services, called a ZoneID, in case of Mansion). EndpointIDs provide a self-certifying name which provides an AOS authentication mechanism, but leaves key-management (or rather, ScID management) to higher layers.

Another advantage of using ScIDs is that they are fixed-length, independent of the public key-length (and much smaller), and that the corresponding certificate is obtained only at connection time (from the peer) and checked only then. New certificates can be generated (with fresh expiration date, for example) at any time, without invalidating the ScID, which only hashes the public key, not the full certificate.

Note that because ScIDs are self-certifying, there is no need for an external, trusted, binding between a key and a name, such as needed in e.g., x509.

6. The ScID is returned to AppMW as part of a listen endpoint address (struct endp) when AppMW creates an endpoint for communication or prepares for receiving an AC. See the API description in section 5 for details.

Implementation. We wrote a library, zonelib (see http://www.cs.vu.nl/~guido/projects/zonelib/), based on the OpenSSL toolkit, for automatically setting up secure (encrypted, reliable), authenticated channels based on EndpointIDs using the TLS protocol. Alternative implementations are possible, the basic principle is that the client requests the public key certificate from the peer, checks if that public key matches the ScID and then does an RSA authentication and key-exchange protocol, (challenge-response) to set up a secret key which is used to protect the communication channel after authentication succeeded.

The target AOS should present a self-signed certificate to the initiator of the connection request. The self-signed certificate's public key is hashed by the 'client' zonelib and the ScID is returned to AOS for (possible) comparison with the ScID it expects. The server asks for the same information (i.e., source public key certificate) if bidirectional authentication is required. An RSA authentication and key exchange protocol is invoked based on the exchanged public key(s) to set up a secure, authenticated channel. The steps above are currently implemented using the TLS protocol. The exact location in the protocol where the ScID is compared may differ for different TLS implementations (e.g., the OpenSSL API differs from the Java TLS API in this respect). TLS/SSL (secure sockets layer) is chosen for practical reasons (almost universal support and secure implementation, and in addition it supports multiple ciphers out of the box), but it can be replaced by another protocol if required.

An AppMW may request a set of chosen chiper suites like TLS_DH_RSA_WITH_3DES _EDE_CBC_SHA (a cipher suite or set thereof is also called a security suite, or secsuite, in this document[7]), although we do not mandate any cipher suites to allow for alternative implementations. See appendix A for a list of example cipher suite strings. The cipher suite strings in the protocol specification (see Appendix) are reserved and must correspond to the cipher suites as they are defined in the TLS v1.0 RFC. **[new]** It is possible to extend an AOS with 'home-grown' cipher suites, for example to experiment with new or extended authentication or crypto protocols. These cipher suites should (obviously) be named such that they do not clash with any cipher suite string names that already exists, e.g., such as defined in the TLS RFC. Most SSL/TLS implementations support multiple cipher suites out of the box.

Zonelib (which uses OpenSSL) can be used in AOS for setting up secure connections between authenticated (AOS) endpoints, or can be used as a template for implementing this functionality in AOS. The AOS implementation should be compatible with zonelib (interoperability should be tested), i.e., with SSL/TLS.

2.2.3 Secure Communication

AOS is used to set up protected AOS-to-AOS channels for the purpose of shipping an AC or to facilitate secure AppMW-to-AppMW level communication. Mutually authenticated, encrypted AOS-level channels are set up using an (EndpointID based) authentication and key exchange protocol at the AOS level. Session management and multiplexing connections over a single encrypted AOS-to-AOS connection avoid the expense of key generation for every connection.

An AppMW-level middleware process or service (e.g., naming service) has an *index* (analogous

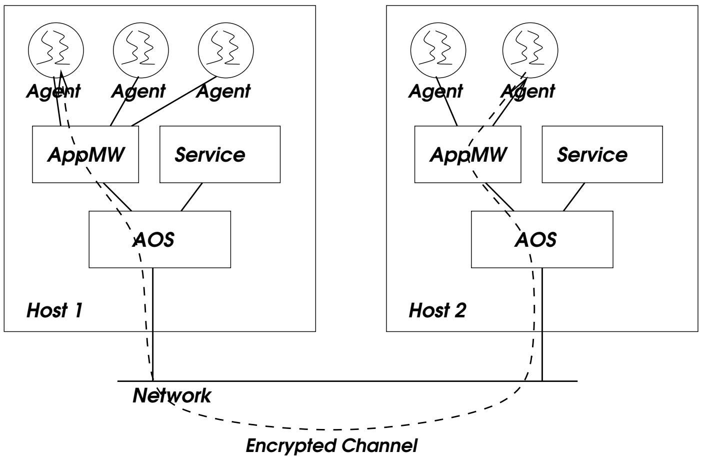
7. Cipher suites strings define what initial handshake authentication mechanism is used, and what (symmetric key) crypto algorithms and hash/MAC functions are used for securing the resulting channel. See the TLS RFC or a security handbook for more information.

to a port) relative to AOS which indicates the specific process that the connection is intended for. Agents run on top of AppMW and may in turn have an index relative to their AppMW level process (fig. 4). In case of agent migration, a hard-to-guess transaction id (xid) is used instead to locate the AppMW for which an AC shipped by AOS is intended.

In Mansion, a separate AppMW-to-AppMW authentication (e.g., EndpointID or zone based authentication to authenticate an AppMW process as a zone) and possibly key-exchange step is made over the AppMW-to-AppMW channel set up by AOS to authenticate the AppMW level process as a zone member, where AOS provides confidentiality using encryption of the base channel between the two AOSes that the two AppMW processes run on[8]. An end-to-end authenticated and encrypted channel can also be set up separately at the AppMW level when end-to-end (AppMW-to-AppMW) confidentiality is required.

Generally, it is sufficient to provide authentication at the AppMW layer and trust AOS to provide confidentiality (base AOS to AOS channel encryption) of the shipped information.

An AppMW should probably implement at least end-to-end authentication on top of AOS connections, for example using ScIDs. A communication channel set up by AOS does not provide end-to-end authentication or privacy (although it does provide confidentiality at the AOS level if requested by AppMW)[9]. ScID-based AppMW authentication is done for authenticating Mansion zones (and possibly AgentScape locations), see [van ’t Noordende et al., Security in a Mobile Agent System, 2004]. AppMWs should probably also at least use a MAC-based scheme to provide end-to-end message authentication and integrity protection, unless security is of no concern. However, the latter is not considered realistic for any distributed (multi-agent) system. AppMW-level (end-to-end) security is considered an AppMW design / implementation issue, and no specific implementation is required or enforced (or even seen) by AOS.



8. Communication (local IPC) between AppMW and AOS is assumed to be trusted (by AppMW, and by inference also by the peer which with communication takes place), and provide confidentiality (operating systems generally provide confidential interprocess communication).

9. Note that if two AppMWs trust each other, and both of those AppMWs trust their AOS and the IPC channel with which they communicate with their AOS kernel, AOS-to-AOS connection confidentiality is probably sufficient. End-to-end authentication is another matter, though.

Figure 2. Communication through AOS

3. Architectural notes

3.1 Dispatchers

AOS supports multiple dispatchers, each supporting a type or RPC. This allows us to remain compatible with the current XML-RPC implementation of the AOS API, but at the same time to support SunRPC (for Mansion / C interoperability). It also allows for future extensions.

A dispatcher is the entry point for calls from a middleware process to AOS. Each call takes a role's cookie as an argument so that the calling process can be securely identified. The dispatcher has a contact point which is accessible only to local processes on the same machine. Each dispatcher implements a server-side stub for RPC calls which exposes the AOS API in that RPC's format. Functionally, all dispatchers implement the same interface (AOS API).

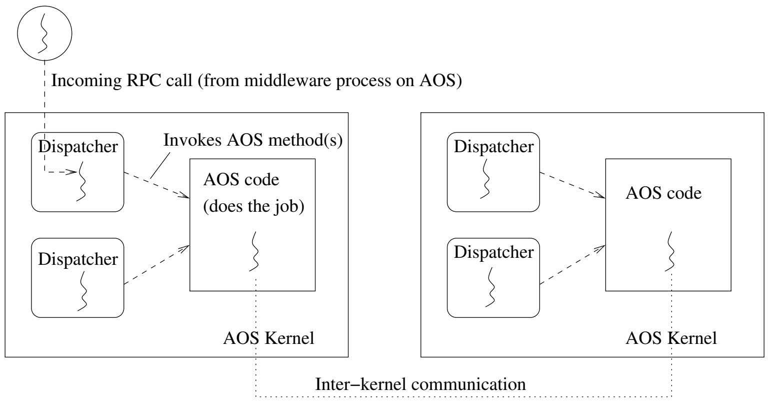


Figure 3. Dispatchers. Figure shows an example of two AOS kernels, each having two dispatchers. Each dispatcher handles invocations for one or more processes. Each dispatcher has at least one thread waiting for RPC invocations, or it might have multiple threads (e.g., one thread per invocation).

Each dispatcher has at least one thread which waits for incoming calls. When the thread is activated (or a new one is created) by an incoming call, the call is decoded (unmarshaled), a check is made if it is allowed by this caller (checking the cookie's rights) and the invocation is made. When the invocation returns, the return value (or possibly error) is returned in properly marshalled format to the caller using the RPC's return message (reply) primitive.

Background Previously, AOS supported only XML-RPC and contained a single main thread which did a select call to see if any data had arrived on a socket. If that was the case, a read was done to obtain the XML-RPC packet (which was shipped over HTTP). The main dispatcher thread was embedded right in the middle of the (then Python) code.

The idea is that the thread is now pulled out of the AOS code and placed in a separate dispatcher 'module'. Dispatcher and main AOS implementation code are now separated in different parts of the code. Pulling the dispatcher thread(s) out of the kernel gives a clean model and allows for multiple dispatcher designs, even for multiple dispatchers for different RPC types (each serving one or more AppMW-level processes) running at the same time.

Separating dispatchers from the main AOS code also makes the kernel design clean: the kernel consists just of the main code and data structures and exports an interface containing the AOS API to the dispatchers, which make calls on it.

The dispatcher threads still drive the code of AOS from the middleware side, as the dispatcher makes the calls on the native AOS implementation as a result of an RPC call. Locking (e.g., using semaphores or mutexes) may be needed to keep different dispatcher threads out of each other's way, either in the main kernel code or in the dispatcher code.

Inside AOS, independent other threads may also exist to handle, for example, multiplexed communication over (secure) sockets. These threads reside at the 'bottom' of AOS.

In principle, there is only one dispatcher for each RPC type. In case of multiple roles / processes using one dispatcher at the same time, particularly when some methods of the AOS API are blocking calls, using multithreaded dispatchers is almost inescapable. In this case, multiple (possibly blocking) RPC calls can be handled concurrently by different dispatcher threads. Note that there may be limits to the number of concurrent requests that can be handled at one time, which can be handled in various ways by different dispatcher implementations.

3.2 On marshaling

Binary formats are much preferred over ASCII representations such as XML for efficiency reasons (see various recent papers, e.g., G. Eisenhauer et al., "Native data representation: an efficient wire format for high-performance distributed computing", IEEE TPDS vol 13 no 12, 2003).

To meet the goals of being reasonable efficient yet to ensure interoperability and language indepencence, we define data representation on the wire using a binary, platform independent format. We use XDR for this, as it is very simple (forcing us to avoid the pitfall of using too complex data types on the wire - we shall have to stick to things that all languages know about such as character arrays, integers, etc). Libraries to convert standard data types from the native representation to XDR are available for many languages, or relatively simple to implement manually.

For this reason, in the remainder of this document, we use XDR to specify data structures.

[new] Some dispatchers or stubs have constraints on the maximum size of the RPC messages they can use. This particularly applies to rpcgen generated stubs and dispatchers which use UDP as transport. A note on this issue is added to the specification in section 5.2.

3.3 A note on AppMW-level services (servers)

Beside the AppMWs itself, many AppMW-level servers or services may run on top of AOS and use the AOS interface for, in particular, communication. An example service that runs on AOS is a location service (i.e., some kind of database server), a Globe object server (GOS), or a web service. Those servers just listen for connections using the AOS API (create_listen_endpoint / accept functions). MW processes can use the connect call to connect securely to servers running at the AppMW level using AOS'es connect call.

Previous AOS API (v3/4) specifications assumed that AOS would do process management. This would have made it possible to put resource control and jailing (see van 't Noordende et al., Security in a Mobile Agent System, 2004) inside AOS. Jailing and process management, however, are now part of the AppMW layer, AOS only knows about roles. This means that either an init process (see below) or AppMW itself has to create roles so that services can use AOS after they are started up.

Init and AppMW's roles generally have access to all methods of AOS, but can only access data structures related to their own role, such as created child roles, listen endpoints, open communication channels, and Agent Containers (ACs). Roles may never create child roles that exceed their own capabilities. Server roles typically have only a subset of the capabilities that init or an AppMW has. For example, most services will only be able to invoke communication related methods of AOS.

3.4 The Init Role

When AOS is first started, it creates an initial role and associated cookie, the **init role.** This role is allowed access to all methods of the AOS API and is the root of AOSes role tree. Typically, some kind of init process (see below) obtains the init role's cookie and creates new roles for the processes it starts up.

The init role is the parent of all roles. Processes are managed outside view of AOS. A role is passed to a process by the role's parent (which itself must have a role that allows it to create a new role), to enable the process to do the role's allowed operations and access the role's resources. A process can freely pass cookies around (i.e., assign or share a role) to any other process or processes. It is assumed that a process takes care to only pass a role (cookie) to a process it trusts. Note that for secure operation, a process should not be given a role including methods that it does not need to invoke. This to minimize damage when the role is guessed (or the role-holding process is taken over) by an attacker.

Init is not directly the parent or owner of all roles that use AOS. A role only has direct knowledge of its own child roles (and the processes that own those roles). Grandchildren are known only to the role's children, and so forth. However, when a role decides to revoke (delete) a role, all subroles of that role and the resources (e.g., ACs, communication endpoints, open connections, and their subroles.) they hold are also deleted[10]. This property of deleting a role holds for every role. Otherwise, cleaning up role state would become very difficult due to incomplete knowledge about the roles in the role tree.

3.5 Using Roles

Init has permission to access all methods of the AOS API and set the roles for the AppMW processes it creates The created AppMW or Init itself can then start up additional AppMW level processes (e.g., services) using the same role or another one.

10. When a process that holds a role dies, but its child roles have to stay alive (for example), the parent should refrain from deleting the dead process'es role until it is safe to delete this role. This kind of coordination among processes holding roles has to take place at the AppMW level, as process management is part of that layer.

The role model exist to limit processes to access only those methods and resources that it needs to access, which limits the scope of an attack in case a process is compromised. Resources such as ACs, communication endpoints, open connections, and child roles, are always tied to a role, which ensures that only processes with that role can access those resources. AOS can also place limits per role on the resources (e.g., cumulative disk space taken up by an AppMW's ACs) that roles can use, or per role subtree.

A role can only be deleted by the parent role of that role (or indirectly when the parent's role is recursively cleaned up). Normally, a process (role) would create a role for each created process, which then does its thing with this role, such as waiting for ACs, listening for connections, etc. However, roles can be used more flexible than this.

[new] As an example demonstrating the flexibility of roles, consider the following scenario. A particular middleware system (AppMW) waits for ACs, and passes these ACs to an (AppMW-level) agent server after it verified and accepted the agent. AppMW also makes a selection of which agent server to use after it inspected the incoming AC (based on for example the language in which the agent was written, or perhaps some quality-of-service agreement with the agent's owner). AppMW can create a new role R which enables only the methods relating to AC access (the role is eventually passed to an agent server, this specific role only permits the agent server to only access ACs). Then, AppMW waits for an AC using the wait_ac primitive (see API), using role R's cookie (see API description in sec. 5). If the AC arrives, and the TOC checked out, AppMW can inspect the content of the AC. Note that the AC is only accessible to processes that have the owning role R's cookie.

If there is some error with the AC, AppMW reports an error back to the sending AppMW and deletes the AC from AOS. If the AC checked out to be OK, AppMW finishes the AppMW-level Agent Transfer Protocol (ATP) by informing the sending AppMW that it accepts the agent. Next, AppMW can pass role R's cookie and the agent container ID (ACID) of the newly arrived AC to an AppMW-level agent server of its choosing. This agent server can then fetch the agent's code from the AC (using role R's cookie), and start the agent up. Note that this is just one example. Several other usage scenario's are conceivable, depending on the way in which AppMW is structured.

AppMW, agent server and other AppMW-level processes can hold multiple roles (cookies) and pass them around to other processes at will. Proceeding on the scenario above, it may for example be that, after it is started up by an agent server, the agent interacts directly with AppMW's main process rather than the agent server in order to make changes to its AC (AppMW's main process can use role R's cookie also), while the agent server only uses the AC to start up the agent from the agent's code segments. In this case, both AppMW and the agent server use the same role R to acces the AC.

Several different variations on sharing or passing a cookie among different AppMW-level processes can be conceived using the role model. As shown, the role model is rather flexible, and usage is completely determined by the AppMW layer.

3.6 The Init Process

Every AOS/AppMW configuration needs and uses some form of init process, which is the process that holds the init role. Init may come with different names in different systems, for example, AgentScape calls it the Host Manager process, whereas Mansion currently names it Init. These are just different names for processes with similar functionality; in the remainder of this text we call this init-role owning process simply init or the init process. As far as AOS is concerned, these processes or their exact implementation are not visible: AOS knows only about roles.

Init can be viewed from the outside similar as a UNIX init process, or shell, which spawns the initial processes and is the root of the process tree, under init. As far as AOS is concerned, only the init role exists, and the mapping of this (and other) roles to processes in a process tree is transparent to AOS. Init is started up before any 'regular' AppMW processes are started up.

Init is simply an AppMW with that distinction that it is the first and can still do everything on its kernel, including limiting the power of the roles that it creates, because it owns the init role. Init creates roles for other processes using AOS and starts up these processes. Typically, init will create at least one AppMW process (e.g., the main Mansion or AgentScape middleware process), which then creates other processes like AppMW-level services, agent servers, etc. Or, in some cases, Init may be used to start up a collection of processes that together compose one or more initial middleware configurations.

3.7 Using Init for Host Protection in Mansion

[new] This section is highly tentative and only an example of some security mechanisms that may be applied in a UNIX environment[11]. We (Mansion developers) are currently working on security mechanisms (i.e., user-level jailing mechanisms) that should be usable in any mobile agent system as a general protection tool, irrespective of the underlying UNIX operating system. The description below may or may not apply to a specific implementation or system and is subject to change. [12]

For host protection, given that this funcionality exists on the local OS, init can create a chrooted environment for each of the AppMW processes it creates. Chrooting (change-rooting) confines disk access to a private part of the disk (e.g., /tmp/), which is visible as the root directory to the chrooted process (i.e., it can not see, let alone access, the parent directory of its chroot directory). In addition, scheduling parameters (i.e., 'nice' or some low priority) and the process'es effective UID can be set by init prior to execution so that the damage that the process can do is minimized[13].

The disadvantage of using chroot is that the chroot system call requires root privilege. This

12. We do not aim to protect windows based systems, as these do not provide the necessary support for process tracing, and differ too much from UNIX systems to feasibly support comparable process jailing mechanisms. Windows systems should probably restrict themselves to accepting only Java-based agents, or other languages that provide runtime protection through their runtime system (e.g., Safe-TCL).

13. Jailing could be used instead of chrooting to 'confine' an AppMW process. However, jailing is specifically considered useful for protection of a machine or middleware against (malicious) agents, which are less trusted and are allowed access to fewer resourcess than AppMW level processes; moreover, a jailed process cannot itself become a jailer, so jailing can only be applied once within a tree of processes (in Adam Balogh's jailing implementation). Therefore, chrooting, (UNIX) UID based protection and scheduling parameter based protection (which remains in place for any child process created by the protected process) is probably more suitable when implemented in init. Other jailing implementations exists (for Linux and FreeBSD) which may be more flexible than the described implementation.

implies that init must be run by root. However, if host protection is an issue, the system administrator (root) will likely be involved in deploying AOS / init, which can be used / secured by the system administrator to enforce the host's security policies.

If the started-up middleware requires so, it can implement additional chrooting[14] or jailing for the agents, agent servers, or services that it itself starts up, in addition to limiting the power of the AOS roles they have. This can be done either by setting up an OS jailer, or by configuring the agent's runtime system or interpreter. It is important to note that neither of these actions should be able to bypass the basic 'chroot jail' (or other protection measures) set up by init. Thus, although an AppMW is assumed to take protective measures to protect the machine from malicious agents or service processes it started up, there is always a second line of defence to protect a host from a malicious, faulty, or badly programmed AppMW that fails to put up such defences.

By change-rooting (or at least by executing as a harmless user, managing disk quota) the initial AppMW (or maybe AOS/init as a whole) processes, the host can be protected against certain actions of AppMW or any of the processes (including agents) that these AppMWs created.

What about all this Host and Resource Protection Stuff?

Jailing or a similar protection mechansim (e.g., user-level sandboxing) can be used within AppMW in order to be able to protect the middleware and its resources (e.g., cookies, diskspace, etc.) from malicious or buggy agents that attempt to find information about other agents, the middleware, or the machine and/or attempt to destruct, corrupt, modify or steal data or consume all available resources, like CPU cycles, memory space, or disk space, in order to bring the system or parts of it to a halt. However, note that a jailed process cannot jail its own children using the linux jailing system call. This is due to a restriction in the jailing implementation (see Balogh, 2003)[15]. For that reason, AOS/init and/or the AppMWs started up by init are chrooted and possibly executed using different user-IDs, rather than jailed. This way, AppMW can still jail its own children (e.g., agent servers), but the host is not completely dependent on this protection. Rather, jailing and user-level sandboxing are used to protect the middleware itself from buggy agents (and possibly to confine agents, see van 't Noordende et al., Security in a Mobile Agent System, 2004). If AppMW does not require OS-level jailing, for example because it only starts up Java agents, AppMW or init and the processes that it creates can be jailed as a whole rather than chrooted, to further protect the machine. However, we assume for the initial Mansion prototype that init provides a chroot environment, and AppMW (Mansion MW) uses OS-level jailing (if available) or user-level sandboxing to protect itself and the machine agains the (binary) agents that it starts up.

Resource protection / limitation (e.g., CPU and memory usage) is not considered in the current design, although some measures can be taken to protect the system against too many resources

14. Note that the behaviour of executing chroot multiple times is unclear to the author at the time of this writing, and may differ from platform to platform. Therefore, init should probably be the location for enforcing chroot policies for host protection, and chroot may or may not be usable in AppMW after using it in init. Linux jailing (A. Balogh, 2003) can also implement a chroot jail independently of chroot by argument subsitution, and does do relative to any possible chroot jail set using the chroot call. Note also that a host owner may start up AOS and init in a chroot jail as an alternative to doing this inside init.

15. We need to figure out how this is in FreeBSD (Niels Provos') or other jailing implementations.

being used (e.g., using a lower scheduling priority than AppMW's for started-up agents if possible).

3.8 Runtime Initialization of AppMW and Other Processes

AOS writes the initial role's cookie in a protected file[16] where it can be read by the first process, init or the host manager program. The file is written in the protected AOS directory together with other protected data (e.g., key store). Except for init, no processes should be able to access this directory (AppMWs are chrooted to a different directory, and optionally started under a different userID, to protect AOS directories from being read by AppMW or other processes).

Init should make sure that the AOS RPC / dispatcher transport address (e.g., local UDP or TCP port or unix domain socket) is passed to the AppMW in a secure way, in addition to its role's cookie. **[new]** Cookie should be kept confidential, so security implies confidentiality here. This can be done using a readable (to AppMW only) file in the AppMW directory, or through IPC (e.g., a unix pipe between the parent and child process). Note that environment variables are probably not sufficiently secure, as they can be retrieved using 'ps' like utilities by any user on the same machine. Note that the client's (AppMW's) RPC type is programmed into the AppMW code, or linked in as part of the RPC stub linked into that code. The dispatcher type and transport endpoint suitable for that AppMW is selected by init or some other way external to AOS.

3.9 Note on File with AppMW Cookie and Dispatcher Endpoints [new]

This is the way in which Arno writes cookie and transport related information on disk. Note that it is not defined where this file is written: filename information is passed as an argument to AppMW and AOS at startup time; security is up to the script / user or init process which creates the directory structure for AppMW and creates its role and possibly dispatcher (endpoint). Note also that the SunRPC protocol entry probably needs a discriminator for the transport type (UDP, TCP or some form of IPC mechanism) it uses.

Provided as 'specification by implementation'.

```
 <kernelcontact>
       <cookie>...base32...</cookie>
       <contact>
               <protocol>XML-RPC</protocol>
               <port>22001</port>
       </contact>
       <contact>
               <protocol>Sun-RPC</protocol>
               <port>22002</port>
       </contact>
</kernelcontact>
```

16. In earlier versions of this document, when an AppMW-level process was started up, cookies and the like were being passed to it via an environment variable or argument by the process that created it. However, such variables are often readable using 'ps' like commands, and thus insecure. Using a file to pass this information, given that the AOS administrator / owner makes sure that the AOS or AppMW directories are only readable to AOS/init or the intended AppMW, is probably sufficiently secure. Using a pipe may be a simpler and even more secure mechanism.

3.10 Note on the native (internal) AOS interface

The RPC / dispatcher (general API) interface does not show all calls on the native interface interface used by the dispatchers. I conceive at least the following call on the native API to be useful:

[new] Removed allowed() call. AOS simply returns METHOD_NOT_ALLOWED if not allowed. This avoids atomicity failure.

```
/* Set_parameter can be used to set variables of AOS at runtime.
 * Generally, parameters are fixed and compiled-in limits (a user of
 * AOS can query those parameters using the get_parm() call on the AOS
 * interface). However, sometimes parameters may have to be set when AOS is
 * started up, for example for debugging purposes. A call like this
 * may be usable for that. It can also be used at kernel initialization
 * time by a separate 'startup' module in the AOS kernel code.
 */
set_parm(cookie_t c, [cookie_t role_c], char* parameter_name, int value);
```

[new] Cookie c is necessary when this is not a private method only callable by the user that started up AOS, but when it is callable on the API by init or some other process that holds the init or an administrative role. We probably should include this method in the API in v6, Obviously, we (AppMW designers, the init process that creates the top-level AppMW roles) have to make sure that only administrative roles have the set_parm() bit enabled in their role bitmap. We may require role_c to be able to set *role-specific* parameters (like disk quota's, etc.) See also the description of get_parm() in section 5 for info on this issue (todo v6).

For now, we leave it up to the AOS implementor to somehow include this method, or perhaps to just read in all parameters from a startup or header file. Marked todo for v6.

4. AOS API. Introduction and Data Structures

The AOS API is discussed in detail in this section. First we introduce the basic definitions needed for the API. When possible, we describe the function / use of the definition immediately. In some cases, the description only becomes clear after reading the method descriptions using them. As this document is intended as an implementation reference, we assume that this approach is acceptable. An overview of the API is given in section 5, followed by a per-method discussion.

4.1 AOS - Basic type definitions

Basic type definitions. All definitions are given in XDR. Note that XDR also defines the ranges unless explicitly specified differently. For example 'int' is a signed int32 (see XDR RFC).

```
const AOS_PROTO_VERSION 5;


typedef unsigned char sha1[20];
typedef sha1 scid_t;
```

RFC 3174 specifies sha-1 output as 5 32-bit words, but in practice every hashing implementation returns a char array in platform independent ordering, (big endian). Thus, I changed the definition from int[5] in AOSAPIv3 to char[20] (since v4). A sha-1 hash consisting of only zeroes is forbidden in this API. Note that higher level middlewares may encode this using base32, which is a valid ScID encoding. Base32 encoding of a binary ScID leads to a 32 byte output (no null-termination):

```
typedef char sha1_enc[32]; /* base32 encoded sha-1. */
typedef sha1_enc scid_b32_t;
```

Roles

Each process using AOS must have a role. A role is a combinarion of a 'cookie' (an authentication token) and a bitmap specifying which methods are allowed by that cookie. The bitmap is stored in the AOS kernel when it creates the role. Each role is a bitmap, with a bit for each method. bit equals 0 means not allowed, bit equals 1 means allowed. When a role creates a new role, it may never exceed its own role, i.e., the new role is accepted iff:

`new_role == (new_role & parent_role);` where & means bitwise AND.

A middleware or init process can define a few role bitmaps for commonly used 'role classes'. Of course, any bitmap may be passed, these are just defined here as an example. The bits in the bitmap are ordered from left to right, i.e., the leftmost bit is the first method of the API (see section 5). Currently, role_t is 4 bytes, with the rightmost (unused) bits left 0. For possible future extensions, we define a type role_t:

```
typedef unsigned char role_t[4]; /* the role bitmap */
typedef int roleid_t;   /* the roleID */
```

Example role bitmaps are (note todo fix to correct bits):

```
INIT_ROLE  11111111111111111111111100000000 /* allow all API calls */
APPMW_ROLE 11111111111111111111111100000000 /* allow all calls */
SVC_ROLE   00000000000000000111111100000000 /* only communication */
```

Note that an agent has no role with respect to AOS. In principle, an agent may never invoke methods directly on AOS, as all its calls should go through AppMW using the AppMW's interface (API). Other middlewares may make a different decision, though (e.g., to allow direct access to ACs).

Each role has a cookie, which is a secret known by the kernel, the role itself, and the role's parent (process).

```
typedef char cookie_t[16];
```

Cookie must be hard to guess. Although AOS only allows RPC calls from the local machine, even there could be a lot of imposters or malicious programs seeking access to AOS'es methods to mess up the state of AOS and/or its users (middleware or agents). For that reason, we define cookie as a 128 bits number. AOS creates it to make sure it is unique and really random, and then passes it (as a return value) to the role's creator who passes it to the process(es) acting in that role. The kernel stores the cookie and the associated rights infinitely, until explicitly deleted, even after restart. Note that not all resources held by a role survive a crash: ports and connection endpoints / identifiers are released after a crash and need to be set up again (these ports / identifiers are considered transient). Finalized ACs are kept (for some period, see sec. 5.3) after an AOS crash or system reboot.

Note that with connectionless IPC/RPC between MW and AOS, MW may well be unaware that AOS crashed and may (succesfully) attempt to use AOS after it restarted. AOS returns an error when AppMW attempts to use AOS after a crash using a special error code (returned as a result of any method on the AOS API), which informs AppMW that all its transient resources are lost. Similarly, an error is returned when a deleted role is used. When a process attempts to access a deleted role, it is informed that AOS does not know the role (cookie). An example way to handle such an error is for the process to kill itself (exit), unless it holds cookies for other roles than the deleted one.

When a cookie is presented as part of an invocation, AOS looks up if the invocation of this particular method is allowed by the role bitmap registered with that cookie.

```
enum segmode_t { TRANSIENT=0, PERSISTENT=1 };
const SSZ 256; /* string size */
```

SSZ defines the size of the subtype and description strings in the TOC. It is changed from 128 to 256 bytes in AOSAPIv5. For now, it seems unwise to be short on space.

The array sizes inside the toc-entries are statically defined to allow for efficient lookups inside the TOC. Using SSZ and the toc_entry struct below, the struct for the TOC entries is fixed size.

Description of segment types.

Segtype_t is used in the struct toc_entry which is described later in this text.

```
enum segtype_t {
 ZERO,         /* Reserved. Used for non-existent (deleted) segs */
 TOC,          /* e.g., to store old_toc. User: AppMW. */
 SIGNATURE,    /* Contains signature (PEM). User AppMW (e.g., audit) */
 CERTIFICATE, /* Contains PEM-encoded cert. User: AppMW */
 CODE,         /* Contains agent code. User: AppMW (e.g., Agt. Server) */
 LIBRARY,      /* Contains library code. User: AppMW */
 AGT_DATA,     /* Contains raw data (binary). User: AppMW */
 AGT_PASSPORT /* Segment contains AP. User: AppMW */
 AOS_SYS,      /* 'System' settings (e.g., leases). User: AOS */
 APP_SYS,      /*  Idem: system segment used by AppMW */
};
```

TOC[17] [18] Segments of type TOC contain a table of contents (TOC) segment. This type is useful not only to mark segment 0, but particularly for distinguishing 'old' tocs. Old TOCs are stored by AppMW in order to establish an audit trail. The description field can indicate which toc (it's sequence number) is stored in the TOC segment. Subtype might indicate "old_toc" to indicate it is not the most recent one, but this is rather redundant, as any TOC segment that doed not have segmentID 0 is by definition an old TOC. **[new]** As a (suggested) convention for AppMW, description could indicate "prev_toc: <SegID>" to facilitate tracing back an audit trail in the AC.

Certificate A certificate segment contains a certificate. Subtype can indicate the type of certificate, usually "PEM_x509v3"[19] or something similar. Certificates store the public key of an entity (e.g., the agent's owner or some middleware's public key which was used for signing data) and associated information such as KeyID or the common name (x509 CN field) of the principal owning the public key.

A certificate can also contain a certificate chain, although these then should be embedded into a single file. I am not completely sure how this is done, but I believe the number of certificates that a PEM file can hold is unrestricted. If it is not possible directly, each certificate in the chain should be stored in a separate certificate segment with description containing sufficient information (e.g., keyIDs) to trace back the certificate chain to the root of the chain.

[new] The description of a segment that depends on a certificate (chain), for example a signature segment or an encrypted (AGT_DATA) segment, should contain sufficient information (e.g., the SegmentID or KeyID) in its name or description to help AppMW find the certificate segment, or the first certificate of the certificate chain which contains the public key used for generating the segment. In some cases (e.g., a PEM signature), a KeyID may be already embedded in the signature segment itself which helps as an extra check to establish that the right certificate

17. Note, the protocols.x file (Appendix B) has SEGMENT_TYPE_ prepended to each type. That's just naming of course, for brevity I retain the names without the preamble here.

18. Changed in v5. Type ENCRYPTED_DATA is removed. Encryption is now considered orthogonal to the segment's type and is indicated by a subtype string, see section 4.3.

19. Note, syntax and semantics of (standard) subtype and description strings will be described in a separate document. These strings are bound to change over time while developing and implementing AOS and AppMWs. Therefore, these strings are best defined in an external document.

segment is used to verify the signature (to avoid failure during signature verification or decryption).

Note that a 'KeyID' as stored in a certificate or signature may not always match the KeyID used in an (AppMW-generated) description in the toc entry, since different digest algorithms may be used to generate the IDs. It is up to AppMW to figure out this distinction. By convention, however, AppMW should (preferrably) use SHA-1 to create KeyIDs or other digests, as these are also used to create ScIDs such as OwnerIDs (see sec. 2.2). The way in which a ScID is generated corresponds to the way in which a Subject Key Identifier in a x509v3 certificate is generated, which facilitates comparison of those identifiers. However, since we cannot (and do not want to) enforce usage of a particular digest algorithm on AppMW designers, so care should be taken that the digest algorithm can be found out by the user (AppMW) if ambiguity may occur at the AppMW level.

Signature. A signature segment contains the signature (typically in PEM format) over a segment. Subtype should indicate the signature's type and encoding (e.g., PEM_SHA-1_with_RSA or something similar, use TLS/SSL spec if possible).

Description should indicate which segment this signature is of. Description should also contain the KeyID of the public key with which the signature can be verified (this KeyID should then also be part of the description of the certificate segment containing the public key), or alternatively the SegmentID of the segment containing that public key's certificate can be embedded in the description (KeyID is preferrable as it is more generic and because it can also be used if public key certificates are not stored in the AC but in some external database).

Code. A code segment contains code of an agent. Each agent has at least one code segment. If it has more than one, the different code segments should generally be of a different language. Functionally, each code segment should be the same. Only one of the code segments is to be executed (by AppMW) at a time, and the middleware may in principle decide which code segment to execute if there is more than one available.

A code segment's subtype contains an (ASCII) string indicating the language of the agent. This string should be understandable by AppMW, and may even differ from AppMW to AppMW. For example, a given AppMW may use a special purpose (hand-tailored) JVM which requires modified byte code. Not all code subtypes are supported on all platforms (which is one of the reasons for allowing multiple code segments on each AC), for example binary code segments can only be run on a machine that supports that binary format. Example code subtypes are: java_1.3_bytecode or linux_x86_elf_binary.

Library. Library segments contain libraries needed by one or more code segments. Library segments are supported to facilitate development of agents that use shared libraries, so that these need not be statically linked. In addition, it can save on the execution memory footprint if multiple agents with the same library are executed at the same time. Subtype describes the library type, for example linux_x86_elf_so. or solaris_sparc_a. Description describes the code segment(s) that the library belongs to (e.g., "belongs_to_code_segment: <SegID>")[20]. A middleware first looks for libraries in the AC and places them in a location where the dynamic

20. Various common (mime-type like) strings like these which can be used in the description and subtype fields of code segments, libraries, certificates, etc., will be described in a separate document.

loader/linker can find them. Then AppMW attempts to start up the agent from its code segment, which then loads the shared libraries dynamically, e.g., from /usr/lib relative to the agent's chroot directory (see sec 3.8). Library segments contain binary code and should be checksummed and signed by the library's author and the agent's owner in the agent's agent passport (see below), just like regular code segments. Also, jailing the agent is a good idea when binary libraries are used, even if the agent itself is not a binary agent. Each code segment may use multiple library segments.

Agt_data. An agt_data segment is a segment intended for storing an agent's own data, such as collected information and descriptions thereof. Description is used to store the data's name, set by the agent, in general. Subtype may either be set by the agent or by the middleware, this is the middleware's decision. In principle, subtype is also intended for the agent so that it can classify information that it stored by labeling its AGT_DATA segments using different subtypes, although in some agent systems AppMW may be responsible for subtyping instead, providing different types of data storage to an agent. For example, an AppMW may store data that it encrypted for its agents in a specially subtyped AGT_DATA segment, similarly for signed data. Another example is a subtype for automatically saved agent state in a middleware system that implements a form of strong migration for an agent.

Agt_passport. The agt_passport segment contains an Agent Passport (AP). An AP is a collection of signatures of each code (and library) segment by the individual authors of those segments (if applicable), and the agent owner's signature over the collection of these signatures. In addition, the AC may contain an agent's middleware-specific global identifier and a signature by some kind of application / world entrance daemon, for example. The exact format and content of some part of the agent passport (and its encoding) is left up to the AppMW designer (i.e., as a piece of opaque data), as certain aspects therein are application specific. The XDR specification of the AP is given in appendix A (ap.x). Note that the agent passport is intended for consumption by an AppMW and not interpreted by AOS. Therefore, the AP specification in appendix A is not mandatory, a given middleware designer may choose an alternative agent passport format if it wants to.

Aos_sys. AOS sys is a segment type reserved for AOS. AOSes can communicate information about an agent using (persistent or transient) segments in the AC. This may be security information, resource usage or limit information, or whatever. Content, subtypes and description of AOS_SYS segments is left open for later determination, although the AOS documentation should at least define common subtypes and their meaning when they come about, to ensure compatibility of diffent AOSes.
AOS_SYS is the only segment type which has a special semantics with regard to write_seg. AOS_SYS segments are readable by an AppMW using read_seg, so that AppMW can verify the validity of a TOC. However, modification of an AOS_SYS by an AppMW is not allowed: a write_seg, delete_seg or make_persistent call on an AOS_SYS segment will return an error ACCESS_DENIED[21]. This behaviour is independent of the segment's real mode (transient or persistent). This restriction is imposed because AOS is the only layer which may modify or delete an AOS_SYS segment.

App_sys. App-sys is a similar segment as AOS_SYS, but then for AppMW. The definition of the

21. All error codes are prefixed with AOS_ERROR_. For brevity, I remove this prefix from the error codes mentioned in this text.

content, subtype and description of the app_sys segments is left open to the AppMW designer / implementor.

4.2 TOC entries

`toc_entry` is defined as follows:

```
struct toc_entry {
  int SegID;  /* Matches segment's name in zip-file (e.g., [0,->) */
  segtype_t segtype;
  segmode_t mode;     /* PERSISTENT or TRANSIENT */
  date_t creation_date;
  date_t last_modification_date;
  unsigned int size;  /* in bytes */
  char subtype[SSZ];
  char description[SSZ];
  sha1 checksum;      /* checksum of segment (created by finalize) */
}
```

Description of toc_entry fields.

subtype is a string which is relevant to distinguish between different segments of the same type. For example, a code segment may have as subtype "Java_1.3_bytecode" or "linux_x86_elf_binary", which is inspected by the middleware (which is the 'user' of the segment) to choose the suitable startup mechanism for the selected code segment. Subtype like these will be defined in a separate document, possibly extended by an AppMW designer.

Most subtypes, like code subtypes, are to be defined (and used) by either middleware developers and/or Mansion world designers. An exception may be subtypes for an AOS_SYS segment, which is only intended for AOS and AOS developers may define subtypes to distinguish different types of AOS_SYS segments. These are not yet defined.

Another example of subtype usage may be for an encrypted segment, for example an encrypted data segment encrypted using the key of the agent's owner. Here, the hash of the public key used for encryption (called KeyID) can be used to identify the target of the encrypted data. In the case of an agent's owner, KeyID matches the agent's OwnerID (if the hashing algorithm used is the same, see section 2.2 and 4.2, description of certificate segment type). Using KeyID, the middleware or any other party (e.g., a tool used by the agent's owner to decrypt the data delivered by its agent) can decrypt the segment(s) intended for it. In this case, subtype contains the string: "encrypted_with_key_id: <keyID>". Furthermore, a string "encryption_algo: <description_like_RSA>" may be added to the subtype description if required.

description [22] is an additional string field for extra information similar to subtype. However, this field's use is less constrained. The use of the description field is up to the intended user of the segment (type), generally AppMW.

22. CHANGED: the 'name' field in the toc entry is now (v5) removed; the description field is used to store the name of (data) segments.

An example use of description is as a "name" field, particularly for data (AGT_DATA) segments. Agents know about data segments (generally, it is assumed that the middleware will hide all other segments from the agent) and use them to store data in, which they obtain while they are on the road. An agent will want to give its segments a name so that it can later distinguish the segments again. Description is used for that. In addition, the agent can fill in the subtype string to distinguish different 'types' of data for itself if it wants to (if AppMW design allows).

If a middleware wants to have more 'fields' available than only description and subtype, it can partition these two fields in any way it wants for segments that it uses itself. Any kind of formatting within the SSZ bytes of those fields is legal, even using '\0' characters within, as long as the intended recipient of the segment knows its way about it. To allow sufficient space for subdividing the description and subtype fields into separate parts, we made these fields 256 bytes long.

Note: checksums may never be 0, unless segment is removed.

4.3 The toc struct

```
struct toc {
        int toc_version;                        /* allow for changes */
        toc_entry entries<toc_entries>; /* contains all toc entries[23] */
};
```

The TOC is defined as a struct toc. A TOC consists of a set of toc-entries (struct `toc_entry`). Each toc contains all toc_entries of the content of the AC at the time of toc creation (finalization). For emphasis, the toc does not include the segment's content itself. The toc itself is stored in segment 0 of the AC, segment 1 is the signature (segment type SIGNATURE) and segment 2 contains the certificate (segment type CERTIFICATE) of the signer of the TOC. or possibly certificate chain. Of segments 0, 1 and 2, most fields in the toc_entry is zeroed (see toc.x in appendix A). The specific signature and certificate encodings used for the certificate and signature segments are described in the subtype field of the toc_entry of these segments[24].

4.4 On deleted segments.

Each segment is assigned a segmentID by AOS. Normally, this assignment should be done sequentially (i.e., incremental). Thus, normally there is no gap between a new segmentID and the previous one. However, a transient segment may be deleted, causing a hole to occur. It is up to AOS to decide on whether to reuse the now-free segmentID or the next available one after the currently highest segmentID in the AC. If a deleted segment is not (currently) reused, a hole remains in the TOC.

We specify representation of holes in an AC broadly, to allow different implementations to

23. Note that this is equivalent (on the wire) to int toc_entries; toc_entry entries[toc_entries];

24. Patrick made a decision to retain readable copies the old toc, toc signature and toc certificates in segments -3, -2 and -1 respectively, after a finalize in the Java AOS implementation. This is, however, *NOT* part of the official (xdr) TOC specification. Please refer to Appendix A of this document for the latest XDR specification of the TOC. Actual existing SegmentIDs, on the wire, only have positive Segment IDs and 'old' tocs are never stored in this way (they should be stored as legal segments in the AC if required, e.g., to establish an audit trail.)

coexist, both of them valid. This allows for certain optimizations (discussed below) which may be required under certain usage scenario's. The two ways of representing a hole (removed segment) in a finalized AC on the wire are as follows:

- We remove the toc_entry, causing a compaction of the AC. In this case, two adjacent toc entries may have segmentIDs that differ by more than one. To look for a given segment, a search in the toc has to be done until segmentID is found. A found toc-entry in one TOC is not guaranteed to have the same location in another one, as inbetween other segments may have been removed. Removing entries wastes no space on toc_entries of removed TOCs, and is the way currently implemented in the Java AOS implementation. To look up a segment of a given ID, the array needs to be linearly traversed and the SegID fields need to be compared with the one searched for. When SegID is higher than the one looked for, searching may stop.
- The segment is removed, but the toc entry is not removed. Rather than removed, all fields of the toc entry are zeroed (to ensure privacy) The type of this segment is now ZERO (0), which means the segment does not exist. Calls to readtoc() by an AppMW should only return tocents for existing (non-zero type) segments. The discussion at this point is about on-the-wire (XDR) representation of the TOC, not about what is visible to users of AOS (deleted segments, obviously, are not visible to users of AOS). The approach here effectively equals segment_id with the location (index) of its toc-entry in the toc, when toc is viewed as an array of tocents. Direct lookup is faster than searching. An advantage of this approach is that comparison of segments of a given ID in different versions of the toc (in an audit trail) can be done by indexing in the toc struct at exactly the same location[25].

By supporting both segmentID-based lookup and direct indexing, we can use direct indexing in general as a straightforward and efficient approach, but we do allow the toc to be compacted, at AOS'es discretion. This may be useful as we don't know how ACs will be used in practice. Suppose that an agent creates thousands of transient segments on some hop and deletes most of them before migrating. Then rather sizeable holes may occur. If we can't compact the tocents of these segments, they can eat up a lot of space, particularly if a copy of each such a large toc is stored in the AC on each hop as part of an audit trail. However, in the normal case it is expected that most segments will be added to the AC as persistent segments, and direct indexing will work without any problem.

In short, AOS can expect, but should not assume, SegmentID to match the corresponding toc_entry's position in the toc.

---

25. Arno notes that on current machines, the overhead of (linear) searching can probably be neglected, particularly compared to crypto operations (i.e., signing). This may be true but I am not sure, as many segments in an AC may have to be looked up, and a signature only has to be created / checked once for each AC. A scenario worked out with Patrick turned out that AC's easily can become hundreds, maybe thousands of segments long (I forgot how we got to this number exactly. But on each hop there is already an addition of at least 3 segments when old tocs are saved as part of an audit trail. Then there is the data found by the agent, self-serialized execution state, AppMW management state (on for example open connections to other agents), etc.). So the search may be considerable. In any case, I prefer simplicity of the lookup algorithm here over saving a few bits of removing toc-entries from the toc. Particularly since we can reuse toc-entries / segmentids of removed segments later for newly generated segments.

4.5 tocent struct for read_toc call

Below the tocent structure is shown. tocent is the standard toc entry that is returned to an AppMW when it calls readtoc. It is comparable to the dirent struct used with unix' readdir call. Tocent contains the same info as toc_entry except for the sha1-hash. The sha1 hash is only used by AOS to verify the content of the AC (when it arrives as part of the wait_ac call) and not visible except by looking directly in segment 0 which contains the last finalized TOC (in XDR encoded format).

```
struct tocent {
  unsigned int segment_id;   /* ID of segment. segID 0 is special. */
  segtype_t type;   /* code, data, .. (see enum segtype_t). */
  segmode_t mode;   /* persistent or transient */
  date_t creation_date;      /* sec. since 1-1-1970 */
  date_t last_modified_date; /* idem */
  unsigned int size;         /* segment's size in bytes. */
  char subtype[SSZ];
  char description [SSZ];

};
```

4.6 AppMW-level Listen Endpoints.

An AppMW endpoint simply specified as AOS'es ip/port number, its ScID (see chapter 1), and an index. All listen contact points for AppMW processes have a non-zero index, which is comparable to a tcp listen port. End-to-end security (authentication, possibly encryption) can be required on top of this scheme. This may be helped by publishing the ScID of the AppMW process along with the endpoint. However, this is a middleware design issue.

In principle, an index can be used as a well-known port, in that a given middleware requests the same index (endpoint) relative to AOS every time it starts up. However, guarantees can not be given that another AppMW process was not there first. In addition, when an AOS starts up (or restarts), its part of the endpoint (e.g., its tcp port but in some cases even its ScID) may change; for that reason, an AppMW may never expect that a newly created endpoint with the same index will be completely the same as an endpoint with the same index that it had before. However, as long as AOS runs, the endpoint remains the same until explicitly deleted or an AOS crash or restart occurs (see sec. 5.3).

AOS should allocate a region at the top of the index range (e.g., by counting down from MAX_INT) for allocating indices when the user does not care (i.e., makes a request for any port by specifying index 0 on create_listen_endpoint). This reserves the lowest range of indices for explicitly requested ports. In addition, any non-listen endpoints which may be created internally in AOS (for example, as temporary 'send' communication ports, see muxp.x in appendix A) should be allocated from the highest region of the index space. Communication endpoints are considered transient and do not survive an AOS kernel shutdown or restart (see section 5).

```
typedef unsigned int index_t;
```

```
struct endp {
  char[16] ipv6-address; /* IPv4 addr is special IPv6 addr 26 */
  short port;
  scid_t scid;     /* ScID of AOS */
  index_t index;  /* index of (communication port of) AppMW. */
};
```

26. A binary IPv4 address is embedded in a binary IPv6 address by prepending the IPv4 address with 80 zero bits, followed by FFFF, and then the 4-byte ipv4 address.

5. AOS API: Description of Methods

In Fig. 2, the AOS interface (v5) is shown in a C-style syntax for briefness. There are some important differences with AOS API v3 and v4. All methods are discussed in detail later. In/out parameters are also discussed later.

Method
`int` **create_role** `(cookie_t c, role_t role-bitmap, cookie_t* child_role_c);`
`int` **delete_role** `(cookie_t c, cookie_t child_cookie);`
`int` **create_ac** `(cookie_t c);`
`int` **prepare_wait_ac** `(cookie_t c, char sec_suites[][], struct endp* actp_endp, xid_t* xid);`
`int` **wait_ac** `cookie_t c, xid_t xidlist[], int listsiz, struct endp* src_endp, xid_t* xid, int block_time);`
`int` **ship_ac** `(cookie_t c, int acid, struct endp* actp_endp, char sec_suites[][], xid_t xid);`
`int` **delete_ac** `(cookie_t c, int acid);`
`int` **read_toc** `(cookie_t c, int acid, int offset, int n, struct tocent t[]);`
`int` **create_seg** `(cookie_t c, int acid, segtype_t type, char subtype[], char description[]);`
`int` **read_seg** `(cookie_t c, int acid, seg_id, int offset, int n, char tobuf[]);`
`int` **write_seg** `(cookie_t c, int acid, int seg_id, int offset, int n, char frombuf[]);`
`int` **delete_seg** `(cookie_t c, int acid, int seg_id);`
`int` **make_persistent** `(cookie_t c, int acid, int seg_id);`
`int` **finalize_ac** `(cookie_t c, int acid);`
`int` **create_listen_endp** `(cookie_t c, unsigned int required_index, char sec_suites[][], struct endp* the_endp);`
`int` **delete_listen_endpoint** `(cookie_t c, int endp_descr);`
`int` **accept** `(cookie_t c, int endp_descr, struct endp* from_endp, bool block);`
`int` **connect** `(cookie_t c, struct endp* target, char sec_suites[][]);`
`int` **send** `(cookie_t c, int connID, int len, char* buf);`
`int` **recv** `(cookie_t c, int connID, int len, char* buf, int block,);`
`int` **peek** `(cookie_t c, int connID, int len, char* buf, int block);`
`int` **close** `(cookie_t c, int connID);`
`int` **select** `(cookie_t c, int setsiz, int read_dsc[], int write_dsc[], int except_dsc[], bool block);`
`int` **get_parm** `(cookie_t c, char parameter_name[]);`
`int` **reenable_role** `(cookie_t c);`

Figure 4. The AOS Interface

5.1 AOS API - Method Descriptions

This section describes the syntax and semantics of each call in the API in detail. In/out parameters are described to facilitate the design of RPC protocols, and clarify the meaning of call-by-reference parameters in the C-style declarations. Note that many calls (like write_seg) take a length, or 'n' parameter. This is intended to indicate to the library that n bytes are to be shipped to the kernel, which is translated by the library to whatever RPC (marshalling) representation required, for example in XDR as an opaque< > data structure (XDR's variable-sized opaque data structure on the wire starts with a big endian integer indicating the lenth of the opaque data that follows). Not all bytes may actually be shipped to the kernel, for example because there are restrictions imposed by the transport used to ship the XDR data, such as the maximum size of a UDP packet. So, the length field in the marshalled data may not match the length specified in the call. In this case, the number of shipped bytes should be returned to AppMW as the return value of the (write_seg/send) call.

By convention, the list of out variables start with the return value. On the wire (in the RPC protocol), the way to marshal returnvalues is in order of appearance: first the return value, then the out arguments in order of appearance. Note, the SunRPC (ONC RPC) specification of the API has not been written yet at the time of this writing.

5.1.1 Role management.

Every call contains an argument cookie (c). Cookie is used as an authentication token. Cookie is a random, hard-to-guess (128 bits) secret which is only known to the caller and its parent. Cookie is passed securely to a process when it is started up, for example through a file or some form of IPC. Earlier versions of this document described passing cookie to a child using an environment variable, but this is insecure, as environment variables can be read using 'ps' or similar utilities.

Cookie is used to avoid that unauthorized processes invoke methods on the AOS API (which may otherwise be easy to do, especially if unconnected RPC is allowed, such as SunRPC over UDP). Cookie is looked up by the kernel to match it to the set of rights of the caller. Note: AOS only accepts local IPC connections, no remote connections. This constraint has to be implemented in the dispatcher.

The call create_role is used to create a new role and associated cookie.

```
int create_role (cookie_t c, role_t role-bitmap, cookie_t* child_c);
```

in: cookie_t c, role_t role-bitmap
out: int [retval], cookie_t child_c

The first role is assumed to be held by some kind of init process (see section 3.6), which obtains the init role's cookie through an external mechanism when it is started up. Typically, AOS is started up first (e.g., by a startup script), as a result of which AOS writes init's cookie and some dispatcher information (contact address for RPCs) in a file. Then, the init process is started up, which reads the file from a known location in order to initialize itself.

Init then creates subsequent (child) roles, which can in turn create other roles using the create_role call. Direct parent-child relations of roles are registered in the kernel. Only a parent can delete its own child roles knowing the child's cookie; when a role is deleted, the child's roles are (recursively) deleted as well.

Roles are, in principle, kept indefinitely, even surviving kernel crashes. However, not all resources associated with a role survive crashes. Particularly, communication endpoints and connections are released after a crash (see section 5.5). If a role and its descendant roles are not used for a long period of time (a week), or when they have not been used since a number of kernel crashes occurred,[27] roles may be expected to be out of use and can be deleted by AOS, for example because the process(es) holding the role could have crashed (see section 5.2).

Create_role takes a role bitmap set by the parent role. It is checked by AOS whether it does not exceed the parent role's own bitmap: (new_role == (parent_role & new_role)) should evaluate true (& meaning bitwise AND). If this is not the case (INVALID_ARGUMENT[28]) or if there is another error (e.g., authentication failed, rpc failure, etc.), an error is returned as a negative return value.

If the bitmap and authorization check out, AOS creates a cookie which results in that cookie being written into the parameter child_c. When create_role is succesful, it returns a roleid, which is a non-secret positive identifier for the created role. This roleid can be used, for example, to list roles and their properties or resources in a kernel, without exposing the role's cookie. This may be useful, for example, for maintenance by an AOS owner, or for debugging purposes. Currently, there is no public method on the AOS API that uses roleid. A roleid has a value > 0. Succes, therefore, is indicated by a returnvalue > 0.

```
int delete_role (cookie_t c, cookie_t child_cookie);
```

in: cookie_t c, cookie_t child_cookie
out: int [retval]

Delete_role deletes the role named by child_cookie, which should be the cookie returned by create_role earlier. Only the role's parent (c) is allowed to delete a role. Authorization failure returns an error ACCESS_DENIED.

Child_cookie is specified rather than child_roleid, which also would have been possible. Using child_cookie avoids that an attacker who managed to get hold of the parent's cookie (e.g., by guessing) can easily delete the parent's child roles[29].

Delete_role also deletes any roles created by the deleted role, plus their associated resources (e.g., AC, endpoints, etc). Subsequent attempts to access a resource or invoke a method using a deleted role returns an UNKNOWN_ROLE error.

Note that deletion of a role should cause any pending requests / threads that use this role to return with an error UNKNOWN_ROLE, or return as if no role deletion has occurred, although possibly

27. Timeout would be sufficient, but after a number of crashes, garbage can build up, so keeping an 'epoch' count of sorts is useful to cover that case and be able to clean up garbage even before the timeout has been reached.

28. All error codes are prefixed with AOS_ERROR_. For brevity, I remove this prefix from the error codes mentioned in the text.

with partial data (e.g., when reading from a connection, the current data is read and then the call exits with that data). In the latter case, an error will be returned the next time the method is called.

Role deletion happens recursively for all child roles. For implementing role deletion and abortion of methods currently using the role, we propose a two-phase role-tree deletion algorithm. Roles are first recursively marked for deletion and then deleted. When a role in the role tree (a role in the role tree is further referred to as a node) is reached, it is marked for deletion. When a node is marked for deletion, it is effectively locked, no methods may enter the kernel using this role: new invocations immediately return an error UNKNOWN_ROLE.

As a possible implementation, each method entering the kernel can first check the status of the role (modulo that it exists at all), and if it is marked for deletion, immediately return an error UNKNOWN_ROLE.

The order in which roles are marked for deletion is unspecified: we suggest to first mark the role (sub)tree for deletion and then do a recursive delete, where each node is only deleted when all method invocations have left the node. It may also be that marking and deletion are done in one step while descending the role (sub)tree. Still-existent roles higher or lower in the recursion tree may or may not be locked yet and may continue functioning until this role itself is marked and deleted by the deletion algorithm. This is safe, as one role may never access data structures relating to other roles (except for calling delete_role), and methods in the kernel should never touch data structures or resources belonging to another role. However, the one-step method probably causes the deletetion algorithm to take longer, as nodes not yet marked for deletion may still serve (possibly blocking) requests and thus may cause node deletion to be delayed.

[fixed some error codes] Whether returning with or without (partial) results is acceptable depends on the semantics of the method in question and in some cases on the specific implementation. For example, already received data over an existing connection should probably be delivered to the application prior to deleting the role (the connection should be closed by returning an error message MUXP_CLOSE_ERROR_ROLE_DELETED to the source AOS, resulting in an ROLE_DELETED error being returned by the send() API call). When reading a segment from an AC, returning immediately with the currently read data is sufficient (the AC will be deleted soon afterward).[30]. Incoming ACTP or communication messages should return an

---

29. For this reason also, a role cannot delete itself. A role's parent is the only one capable of deleting a role. Thus, when a role-holding process exits, it should signal its parent that it exited (or the parent should catch the signal resulting from its child process'es exit, if its process is a direct descendant of itself), or when it simply wants its role revoked it should ask its parent to do so. This is a bit awkward, however note that guessing a role is not inconceivable, given that a ROLE_UNKNOWN error is returned to any process trying to guess a role.

    An important reason for imposing the limitation that a role cannot delete itself, is to protect information of the child roles, which can otherwise be easily removed. Note that guessing a role is not the same as overtaking a role-keeping process. In the latter case, it is not possible to protect the child roles as it may be relatively straightforward for an attacker to obtain all the cookies that a process holds and delete them using delete_role (or worse, use these cookies to obtain (sensitive) data from the role's resources (connections, ACs)).

30. The semantics of read_seg() on an error is specified as: return number of bytes of read data first, then next time call is made return the corresponding error code (in this case UKNOWN_ROLE.) Similar error semantics apply to the recv( ) call. It is undefined whether the deleted role received all data or not prior to being deleted. Higher level protocols over the virtual connection should implement message acknowledgement if this is required.

PEER_ROLE_DELETED error to the source AOS (as part of the internal AOS-to-AOS Agent Container Transfer Protocol (ACTP) protocol messages) immediately, leading to a ROLE_DELETED error returning from the ship_ac call. For new communication or ACTP requests, an error TARGET_UNKNOWN is returned to the connect / ship_ac caller immediately, as a result of corresponding internal AOS-to-AOS protocol messages (see Appendix A).

It is also valid to return all methods with an error UNKNOWN_ROLE immediately, without any partial data; as the exact moment that a delete_role kicks in is unpredictable, the result is equivalent to the earlier described solution which does return partial data. Note that role deletion should preferrably be a fast process, and current method invocations should be returned cleanly as fast as possible (so going through a lot of trouble to first obtain all sent data may not be a very good idea).

An implementation can have all methods that implement the API check regularly while executing, and just prior to returning, on whether the role is marked for deletion, and return then with the corresponding error code (and/or partial data, depending on the semantics of the call). This solution may cause deletion to hang for some period of time, at least as long as the longest-taking method using a role within the role (sub)tree. But by marking all nodes in the tree for deletion at once in a single run, prior to actual deletion, it can be ensured that no new methods can enter the kernel for any of the roles in the tree, as that these methods return immediately with an error.

When all users of a given role (threads using a role) have returned, the role and all its resources (ACs, etc.) can be deleted, including roles (causing recursion, except that these child roles may already have been marked for deletion before), irrespective of whether other roles surrounding it have been deleted or not.

Note that crash recovery may be an issue when deleting a role. The highest-up role, the one that is actually being deleted, should be marked for deletion on disk (and sync( )'ed) immediately as a result of invoking delete_role, before the deletion algorithm proceeds, so that this information is available after a restart which happens during role(tree) deletion. This way, when the kernel is restarted, the role subtree is first checked on whether it contains any roles marked for deletion. If this is the case, the whole subtree can be deleted at startup time. All this (at least marking for deletion of the whole subtree) has to be done prior to allowing any AppMW to do invocations on a restarted kernel. Note that calls that were in the kernel prior to the AOS crash would have received or will receive an IPC; no checks will have to be done for current methods in the role-subtree[31].

Normally, delete_role returns 0 when succesful. A negative returnvalue indicates an error.

5.1.2 AC Management

Agent Container Management

These calls are used to manage AC's. The concepts TOC and AC were explained earlier in this

31. There is a whole issue on how to cleanly restart an AOS kernel in the cases where role information should be kept and in the case where all current information should be removed from the kernel (i.e., clean restart). Init probably has a role in such a kernel shutdown, and/or a special signal may be used. Proper cleanup of all deleted roles may cause slower exit of the AOS kernel and/or a slower restart (see also section 5.3).

document.

```
int create_ac (cookie_t c);
```

in: cookie_t c
out: int [retval]

Create_ac creates a new AC. It takes cookie as an argument. If cookie is not authorized to make the call, an error METHOD_NOT_ALLOWED is returned. Another error may be LIMIT_EXCEEDED when an AOS-defined maximum of created ACs per role is exceeded[32]. If succesful, a positive (>0) ACID (Agent Container ID) is returned which can be used for subsequent calls involving agent containers. ACID is a strictly local identifier, which is not maintained throughout the AC's / agent's travel, and which has no meaning when the AC is migrated to another machine. Locally, however, ACID is stable and remains associated with the same AC as long as the AC is on that machine / AOS.[33]

Note that an AC is a resource which is owned by the role (cookie) that created it. Only processes that posses the owning role/cookie may access or modify the AC or delete it - with the exception of the parent role, which may delete its child roles with all their resources, including indirectly their ACs (this holds for all resources and is not mentioned any further in the remainder of this text.)

[new] Changed the descriptions of prepare_wait_ac, wait_ac and ship_ac a bit, and reordered the sections for readability. The signature of wait_ac has changed.

```
int prepare_wait_ac (cookie_t c, char sec_suites[][], xid_t* xid, struct
endp* actp_endp);
```

in: cookie_t c, char sec_suites[][],
out: int [retval], xid_t xid, struct endp actp_endp

Prepare_wait_ac has to be called prior to calling wait_ac. Prepare_wait_ac's primary function is to create a hard-to-guess, random 128 bits transaction id (xid) which is to be passed to the initiator of ship_ac and then has to be used locally on wait_ac (see wait_ac description). Xid is an out parameter written in xid_t* xid by AOS. In addition, AOS fills in the endpoint struct needed by the caller of ship_ac to initiate the ACTP. The actp_endp is needed by the sending AOS to initiate the actp. Actp_endp contains the ip/port address and scid of AOS, which are used by the ship_ac to contact and authenticate the AOS kernel to ship the AC to. XID and actp_endp are assumed to be shipped to the invoking AppMW in a secure way (i.e., over a secure, authenticated AppMW-to-AppMW channel, see sec. 2.2.3).

---

32. How limits are managed is not defined and an AOS implementation and deployment issue. It may be that quotas are managed per role, rather than per role tree. If bypassing a quota by a role is an issue, it is important not to issue role bitmaps with the create_role bit enabled in this scenario. Quota management is left to the implementation and will probably not be implemented in the v5 AOS kernel (except perhaps for disk quota's set by init or the host administrator).

33. See sec.5.3 and the description of the finalize_ac call for information on what AC is retained in the case of a crash of AOS.

Note that the current AOS version (v5) only allows ACTP to take place over a base channel. Reflecting this decision, the index field in actp_endp is always set to 0 by prepare_wait_ac. In principle, it would be possible to use an index other than 0 for shipping an AC, implying that ACTP messages are sent over a multiplexed communication channel in AOS. However, this is forbidden in AOS API v5.

Security of the ACTP at the target side is provided by xid, not the actp_endp. Actp_endp may actually be reused for multiple actp requests, particularly, an AOS may have a single listen endpoint for all incoming actp requests. XIDs are coupled to the role that created it (each XID is guaranteerd to be issued to one role at a time only). Different ACTP requests are identified using XID, and making XID hard to guess improves security in that it decreases the probability that an external attacker initiates an 'valid' ACTP by guessing a XID.

Note that we do not specify a priori what address information will be in the actp_endp. In fact, a scenario (albeit a bit unrealistic) may be that a different actp_endp is returned for each invocation of prepare_wait_ac, for example each time with a different tcp port. Normally, we'd expect that each AOS has one or at most a few fixed, stable endpoints for (multiple) incoming ACTP requests. This tcp listen port may also be shared with incoming multiplexed communication requests (see Appendix A)[34], depending on the AOS implementation.

The caller of prepare_wait_ac should regard xid and actp_endp as opaque data which it passes to a sending middleware, which uses this data as arguments to the ship_ac call (see below).

Sec_suites is used by the caller to set the allowed cipher suites for the (base) connection over which ACTP is invoked. An error may be returned if the current AOS does not support any of the requested cipher suites (NO_KNOWN_SECSUITE).

It is useful, in general, to publish the sec_suite along with the endpoint in some AppMW-level database, so that a connecting party knows what sec_suites to choose from before attempting to connect to the listen endpoint. Alternatively, the caller can also pass the sec_suite to the sending middleware together with xid and the endpoint struct, to inform it of the sec_suites it can choose from when shipping the AC. However, the AOS implementation is able to determine at connect time if any given client / server AppMW sec_suite combination gives a valid sec_suite to encrypt the (base) connection with, and otherwise returns an error to the shipping party.

Normally, prepare_wait_ac returns a value 0 when succesful. A negative returnvalue indicates an error.

```
int ship_ac (cookie_t c, int acid, struct endp* target, char sec_suites[][], xid_t xid);
```

in: cookie_t c, int acid, struct endp target, struct sec_suites[][], xid_t xid
out: int [retval]

---

34. Something to keep in mind is that AOS should preferrably be usable behind a firewall or NAT router. Enabling one or two routes through a firewall or NAT router may be feasible, but allowing large numbers of (dynamically allocated) ports for ACTP or communication channels through a firewall or NAT router (implying the configuration of new (default) routes through the firewall or router) is out of the question.

Ship_ac ships AC acid to the target indicated in struct endp target. The endp struct endpoint is needed by AOS to set up a secure (base) channel to initiate the AOS-level Agent Container Transport Protocol (ACTP). In addition, a random transactionID (XID, see prepare_wait_ac), is provided by the target AppMW to the invoking AppMW, which passed to AOS as an in parameter of the ship_ac call.

XID is required for the ACTP to succeed succesfully. An ACTP can only be initiated if the target side recognized XID as a XID that was generated by an AppMW using prepare_wait_ac. At the target side, an AppMW can collect the AC after shipment using the call wait_ac, using the same XID as the sender used. Only the role that created the XID using prepare_wait_ac can collect the AC. Note that XID explicitly allows for concurrent sends from one AOS to another over the same base channel using different XIDs. The ACTP described in Appendix A supports this as wel as interleaving of ACTP protocol messages with multiplexed communication (MUXP) protocol messages (see Appendix A).

Acceptance of an AC by an AppMW has to be negotiated at the AppMW level after ship_ac / wait_ac. The semantics of ship/wait_ac essentially are that a copy of the (finalized) AC is shipped to a target AOS/MW, and then after that the two middlewares that each have a copy of the AC must figure out which one of the two MWs has to delete the AC, and which one has to (re)start the agent from that AC (note, there is no explicit accept_ac or reject_ac call).

The reason for this semantics is that AOS should not have to block the AC transport protocol (ACTP) and ship_ac call until the receiving MW is done determining if it accepts the AC, which may take quite a while. AOS should be able to 'forget' about the AC after ship_ac / wait_ac until its AppMW decides to do something with it or delete it.

During ACTP (see actp.x in Appendix A for detailed information on the ACTP protocol messages), the checksums of all segments in the AC are verified using the checksums in the AC's TOC at the target AOS side. In addition, the signature of the AC is verified (as are all checksums) by the target AOS using the public key that is included in a certificate in the AC, which should match the self-certifying ID of the sending AOS obtained by authentication during setup of the base AOS-to-AOS channel over which the ACTP takes place. If the AC does not check out, the target AOS sends an ACTP error message to the sending AOS. An error may also be returned if a secsuite matching error occurred (see sec. 5.3). If verification checks out, the AC is accepted and the ACTP is finished by the target AOS, which signs the TOC and ships the signature and its own self-signed public key certificate back to the sending AOS as a receipt. The signer's key used to sign the receipt must match the ScID in endp target at the sending side. This is checked by the sending AOS kernel using the ScID which it obtained and verified during setup of the secure AOS-to-AOS base channel over which the ACTP takes place.

Ship_ac returns as soon as the target AOS verified and accepted the AC as a valid AC, and returned a valid receipt. If the ACTP fails for some reason, causing ship_ac to return an error (except for error AOS_ERROR_ACTP_RECEIPT, see errorcodes.h in Appendix A), the target AOS is supposed to delete the AC it received (note though that this cannot be verified by the sending AOS in general).

The sending AOS may store the receipt (signature and certificate) and the TOC belonging to it in an internal log for auditing. The receipt is not used by AppMW (AppMW trusts AOS to verify the signature in the receipt, which AOS verifies using the TOC of the finalized AC that it sent (the signature should match the TOC that was sent)); when the ship_ac returns 0, AppMW is satisfied that verification was OK. AppMW may additionally verify a similar signature of the TOC generated at the AppMW level by the sending/receiving AppMWs, respectively, as part of AppMW-level auditing, handoff, etc. However, this is external to AOS.

Ship_ac is a blocking call, which normally blocks until the ACTP is finished. If the specified target (XID) does not exist at the target AOS, ship_ac returns an error TARGET_UNKNOWN. It can also occur that ship_ac is invoked too late. Although the time that an ACTP in progress takes to complete is unrestricted (e.g., it may take place over a very slow link), the initial ACTP request has to arrive within an AOS-defined period of time. This period of time, defined in params.h, is set by the target AOS, and can be queried by the target AppMW using the get_parm() call. If the initial ACTP request has not arrived at the peer within this xid invalidation time, ship_ac returns an error TARGET_UNKNOWN. Obviously, it may take a while before ship_ac returns this error. If the real reason for not succesfully initiating the ACTP is unreachability of the target AOS, a PEER_UNREACHABLE error is returned instead. Another reason may be some transient error at the peer (e.g., out of diskspace). Before XID is invalidated at the peer due to the xid invalidation time, several PEER_UNREACHABLE or PEER_ACTP_TRANSIENT errors may occur (the underlying errors may in some cases be hidden by AOS, which may retry initiating the ACTP several times before returning an error). A ship_ac invocation after the peer invalidated the XID yields a TARGET_UNKNOWN error.

Several other errors may also occur: AC_NOT_FINALIZED if the AC was (locally) not finalized (i.e., when the AC's finalized bit was not set (see finalize_ac)). ACTP_AC_TOO_LARGE when the AC was too large for the peer AOS, and ACTP_PEER_TRANSIENT when the peer AppMW is currently out of diskspace. ACTP_ACTP_AC_FORMAT, ACTP_AC_VERIFY may occur when the peer found an AC format error or when verification failed. ACTP_RECEIPT occurs if a local verification of the ACTP receipt failed. ACTP_PEER_INTERNAL is returned if the peer reported a fatal error. SECSUITE_AGREEMENT is returned if a sec_suite handshake is unsuccesful. CONN_SETUP is returned for some (transient) errors during conneciton setup. XID_IN_USE is returned when the XID is already in use at the peer AOS by another ACTP (this should normally not occur and can indicate an attack in progress or some programming error at the target AppMW (e.g., same XID issued to two AppMWs) or AOS (e.g., it created the same XID for two AppMWs)). These and other error codes are described in errorcodes.h (Appendix B).

Errorcodes.h also describes which error codes lead to an invalidation of the XID at the peer: some errors related to failure of the ACTP protocol invalidate the XID immediately and require a newly generated XID using prepare_wait_ac to start another the AppMW-to-AppMW Agent Transfer Protocol (ATP).

```
int wait_ac (cookie_t c,  xid_t xidlist[], int listsiz, struct endp*
src_endp, xid_t* xid, int block_time);
```

in: cookie_t c, xid_t xidlist[], int block_time
out: int [retval], struct endp src_endp, xid_t xid

Wait_ac is a method which waits for an AC shipped from another AOS using AOS'es ACTP. It is the only method besides create_ac which returns an ACID, which can be used to read, modify, ship, or delete the AC. The return value of wait_ac is a positive ACID (Agent Container ID), a negative error code (see errorcodes.h), or 0, in which case no AC arrived and no error occurred within block_time.

As always, wait_ac takes a cookie as a parameter, which is the cookie of the caller to which the AC will belong after wait_ac returned succesfully.

The xidlist argument contains a list of XIDs whose ACTP completion (received AC) the caller is waiting for. Initiating (ship_ac) and waiting (wait_ac) XIDs have to match for ACTP to be initiated succesfull (see prepare_wait_ac). Xidlist is an unsorted list of XIDs. All XIDs are to be placed at the beginning of xidlist, with no zero XIDs inbetween[35]. Wait_ac returns the first available (fully received and verified) AC to the caller, independent of order of occurrence in xidlist, by returning the arrived AC's ACID. Only one ACID is returned at a time, even though multiple ACs may be ready at that time.

Listsiz indicates the total length of xidlist as the number of XIDs (so not the number of bytes). Listsiz is not visible on the wire, it is only used by the (C) stub to fetch (and marshal) the xidlist from the user-provided buffer. A Java version will probably pass an object containing XIDs to the wait_ac call instead. Xidlist can contain up to AOS_MAX_WAIT_AC_XID_LIST_SIZE xids (see params.h). AppMW may invoke wait_ac multiple times at the same time, provided that the xidlists passed with the invocations are disjunct, i.e., no XIDs may appear in more than one xidlist. This can be used, for example, to wait for more XIDs than the above-mentioned maximum. An error XID_IN_USE is returned when another currently processed invocation of wait_ac already waits for a XID in xidlist[36].

The xid argument of wait_ac is an out parameter in which the XID corresponding to a received AC is placed. The xid argument is also used to write the XID corresponding to an error (in the ACTP) into (see below).

XIDs are coupled to roles internally to AOS: when an ACTP message comes in, it will only be passed to a wait_ac when XID was actually created by the invoking role using prepare_wait_ac[37]. If any unknown XIDs (or XIDs owned by a different role[38]) are found in xidlist, an error UNKNOWN_XID is returned immediately, with xid set to contain the (first found) unknown XID. This error may be returned for multiple subsequent invocations of wait_ac, until no more

35. AOS v6 may accept sparse xidlists (with zero XIDs inbetween) as input.

36. It is currently not allowed to pass a NULL or empty xidlist to wait_ac. Future versions of the API will allow this, with the semantics that this instructs wait_ac to wait for any XID that is created by the invoking role. This is useful for AppMWs that use a single thread for incoming ACTP requests, and possibly for (crash) recovery. However, such a method cannot be concurrently used with other wait_ac invocations, either with or without an explicit xidlist, for various race conditions may then arise. The definition of the precise semantics of invoking wait_ac with a NULL or empty xidlist is postponed to AOS API v6.

37. This limitation prevents other AppMWs on the same machine from guessing XIDs and 'hijacking' ACTPs of other AppMWs by calling wait_ac.

38. Note that ACCESS_DENIED is not returned if XID belongs to another role. We just consider if XID is known to the *invoking* role. The distinction between a XID not belonging to the invoking role and XID not known to any role in AOS should be irrelevant to the caller, so we return UNKNOWN_XID in both cases. Giving an ACCESS_DENIED error potentially gives an attacker (an AppMW on the same machine) leverage to search for XIDs known to AOS. These guessed XIDs can then be used by an attacker to initiate an ACTP, thus denying access to the legitimate user of the XID/ACTP.

unknown XIDs are part of xidlist. The same applies to XID_IN_USE errors.

An out variable is provided as struct endp src_endp, in which AOS which writes the endpoint information of the sending AOS kernel. The sender (src_endp) AOS is automatically authenticated by AOS as part of secure AOS-to-AOS (base) connection setup prior to shipping the AC. AOS also verifies that the signer of the AC is the same as the sender (src_endp's ScID) of the AC[39]. ACTP without sender AOS authentication is not permitted. Thus, the ScID in src_endp corresponds to the (verified) sender of the AC. Furthermore, the sending AOS'es IP address is filled in, if possible (else the field is set to 0). Port and index field of the sending kernel / MW may also be filled in, but these values have no meaning (i.e., it is not possible to connect to the endpoint). The src_endp is purely informative for the receiving MW.

Block_time specifies the time (in seconds) in which wait_ac will wait for an arrived AC. Note that if within this time an ACTP is initiated but has not finished yet, the call will return without reporting receipt of the AC; particularly, it does not wait for completion of the ACTP protocol. Block_time 0 means 'poll'.

Note that wait_ac is the only call on the AOS API which has a timeout rather than a boolean block parameter. This is because all other calls with a block parameter will unblock as a result of *any* incoming message, whereas wait_ac can be invoked to block waiting for a single specific XID; there is no method to deallocate an existing XID. If a XID was specified and the ACTP was not initiated within XID_INVALIDATION_TIME (see params.h), wait_ac will return an error UNKNOWN_XID at the time that the XID is released. This does not apply to initiated (unfinished) ACTPs, however, which may hang indefinitely. Thus, failure of a given client AppMW to finish a specific ACTP may cause wait_ac to block indefinitely, which is considered unacceptable behaviour. For this reason, block_time is introduced. If block_time exceeds without an AC being available and no error occurring, wait_ac returns 0.

The errors XID_IN_USE or UNKNOWN_XID cause wait_ac to return immediately without processing any valid XIDs. Other errors may occur, for example when the initiating AOS could not be authenticated. Also, certain ACTP error may occur (e.g., verification errors). These errors are the same as the ones returned with ship_ac, and are defined in errorcodes.h (see Appendix B). In all these cases, xid contains the XID to which the error applies. Some errors of the ACTP protocol (e.g., verification failure) cause the XID to be invalidated (leading to an UNKNOWN_XID error if used again). In that case, a new prepare_wait_ac has to be done and the AppMW-level ATP has to be restarted using the newly created ACTP endpoint / XID. These XID invalidation events are described in errorcodes.h (Appendix B).

```
int delete_ac (cookie_t c, int acid);
```

39. If required, future versions (>v5) of the AOS spec may decouple sending AOS authentication from matching the signer of the AC. As the signer of the AC can be found out from the certificate in segment 2, and the sender AOS'es ScID can be found in src_endp, the coupling is not strictly necessary. However, it seems that this decoupling may open holes for insecure usage of the AOS-level ACTP, although I can imagine situations where decoupling is useful. For example, it could be that on a network of machine a dedicated AOS is deployed which takes care of shipping ACs which are signed by other AOSes. However, this can also be done by first (internally) shipping the AC to the dedicated AOS (if AppMW-level ATP allows for this). I think the consequences for higher-level protocols depending on secure authenticated AC transport between AOSes should be scrutinized before allowing such a decoupling, and it should not be done unless absolutely necessary.

in: cookie_t c, int acid
out: int [retval]

Delete_ac deletes the named AC from AOS. A negative returnvalue may be returned when an error occurred, e.g., ACID does not exist or when c is not the owner of the AC. ACID is returned to the pool of available ACIDs after deletion.

Segments

```
int read_toc(cookie_t c, int acid, int offset, int n, struct tocent
tocent[]);
```

in: cookie_t c, int acid, int offset, int n
out: int [retval], struct tocent tocent[]

A UNIX 'readdir' like call. There is only one call, seekdir is not necessary, as we use an offset. read_tocent reads n TOC entries starting at entry 'offset', and writes the content of their tocents (as a struct tocent, i.e., without checksums) in buffer tocent[]. If succesful, read_tocent returns the number of read entries. This may be less than n: succes is indicated by a returnvalue >= 0. **[new]** Succesful return of read_toc does not imply that all available (up to n) tocents have been returned[40]. If all tocents up to n must be obtained, read_toc should be invoked multiple times (with offsets and n adapted according to the number of earlier returned tocents) until all n tocents have been obtained or 0 is returned, indicating that no more entries are available.

A negative returnvalue indicates an error. Specifying an offset beyond the last segment simply returns 0, an error is not returned in this case. An error code may be returned when an invalid ACID is specified, for example.

Read_toc's offset indicates the next valid (non-deleted, see sec. 4.5) toc_entry in the TOC, irrespective of its segmentID. Segments in the TOC are always stored in-order, that is a segment with a given segmentID may not have a segment with a lower segmentID located after it. However, it is perfectly valid that there is a difference more than one between segmentIDs in an AC, for example because segments were removed. Holes (empty, deleted toc entries) may also occur in a toc but these are ignored by read_toc. Read_toc's offset value indicates the nth non-empty tocent in the TOC, irrespective of segmentID. Read_toc displays the current (possibly non-finalized) information about the segments.

Note: segments 0-2 contain the TOC, signature and signer's public key, respectively. The toc_entry structs for those segments are included in the real, physical TOC segment but with several fields (e.g., the size field) set to 0 instead of their 'real' value (see toc.x in Appendix A). The tocents describing these segments are recreated on the fly with current information by the AOS kernel when read_toc is called.

As soon as a change is made to a segment, its size field in the *runtime* tocent (which is what is returned by read_toc) is updated to reflect the current size. The same holds for other changes to the tocent struct, such as the persistent mode field. The checksum (in TOC) is not updated until

40. This is due to the fact that the client stub or the dispatcher used by the AppMW process may truncate the read request to accomodate an RPC-specific maximum message size (see sec. 5.2).

finalize, but the checksum is not part of the tocent returned by read_toc anyway. Only after finalize will the real, on-disk AC be updated, including valid toc fields. Note that only finalized ACs can survive crashes. An agent programmer should take care that the agent can be restarted from its finalized state, and an AppMW programmer should make sure that an agent is restarted using the code and information stored in the (last finalized) AC after an AOS crash, unless it informs the agent of the 'your AC is reset to the last finalized state, please restart if necessary' event, or it stores all unfinalized state of the ACs/segments used by agents in some kind of AppMW-level cache or on-disk storage system, thus hiding AOS failures from the agent.

Note that until finalize_ac is called, the content of segments 0-2 remains as it was originally, irrespective of changes to the current (non-finalized) AC. That is, these segments remain as they were when the AC arrived on this AOS through wait_ac, or as they were when finalize_ac last caused a new toc to be created and a new signature to be generated. Segments 0-2 remain unaffected by changes in the runtime tocents as returned by read_toc, unless finalize_ac is called, which really writes the physical TOC, signature and certificate to disk. As an aside, if the AC is a newly created AC (using create_ac), segment 1 and 2 are empty segments, so their size in tocent is 0, as are their creation and modification date and checksum (only their segment type is set in this case), until finalize_ac is called.

AppMW may need access to the raw, on-disk TOC directly, either to (double) check its contents itself, to sign it as part of an AppMW-level handoff protocol, or to copy it into a new segment of type TOC so that an audit trail can be created in the AC [see van 't Noordende et al., Security in a Mobile Agent System, 2004 on how this works]. The full content of the TOC, its signature and the certificate of the signer, can be obtained by reading respectively segment 0, 1 and 2 of the AC using read_seg prior to calling finalize. Note that segments 1 and 2 are the sender AOS'es signature and key, *not* the sending AppMW's signature and key. The latter may be more useful to an AppMW than the sending AOS'es signature and key, but this is external to AOS.

Note that all calls related to the AC are stateless, and that there are no filedescriptors or open, close or seek calls. AOS does not keep track of open files and filepointers, etc. As there may be many agents using AOS at the same time, each acting on possibly many different ACs at approximately the same time, making AC operations statefull could cause significant overhead and complexity. Given the limited amount of open filedescriptors for a given process (AOS), I do not assume that AOS keeps any files open between operations. If performance is an issue, AOS could and probably should manage some kind of (in-memory) cache to handle read and write request more efficiently when possible. Segments need only be written to disk (and sync( )'ed) physically only when a finalize_ac call is made (see below), so the design of such a cache should be straightforward.

```
int create_seg(cookie_t c, int acid, segtype_t type, char subtype[],
char description[]);
```

in: cookie_t c, int acid, type_t type, char[] subtype, char description[]
out: int [retval].

Create_seg creates a segment of type type, with subtype and description as provided by the caller (as strings of max. SSZ=256 bytes long). Create_seg will fail if acid is not a valid AC owned by role c. Then a corresponding error value is returned.

Succes is indicated by a valid segment ID being returned, which is always a value larger than 2, as segments 0, 1 and 2 are reserved for TOC, signature and certificate of the (signed) AC. Create_seg never returns 0, 1 or 2. Other reasons for failure may, for example, be that a disk quota or other limit set by AOS (e.g., a maximum of segments per AC) is exceeded (error LIMIT_EXCEEDED).

```
int read_seg (cookie_t c, int acid, int seg_id, int offset, int n, char
tobuf[]);
```

in: cookie_t c, int acid, int seg_id, int offset, int n
out: int [retval], char tobuf[]

Read_seg reads n bytes starting at offset, if available, into tobuf. The returnvalue indicates how many bytes were read. If offset is beyond the segment's last byte (caller can check segment's current length using the read_toc method[41]), read_seg returns 0. If there are less than n bytes from offset, the available bytes in the segment after offset are copied to tobuf, and the amount of bytes is returned as the return value. The first byte of the segment is at offset 0.

Several failures can cause a negative error, such as giving an invalid acid or seg_id, or an access denied error.

```
int write_seg (cookie_t c, int acid, int seg_id, int offset, int n, char
frombuf[]);
```

in: cookie_t c, int acid, int seg_id, int offset, int n, char[] frombuf
out: int bytes_written (retval)

Write_seg writes n bytes from frombuf to location offset in segment seg_id. Specifying an offset beyond EOS (end of segment) is not allowed: AOS segments may not contain holes. Succesful execution returns the number of bytes written, which may be smaller than match n.

Note that the length argument really is not visible in the actual data which is shipped using RPC. Length is used by the stub to determine the amount of data which is to be shipped (marshaled), and encoded using the RPC's data marshaling method (e.g., XDR, in which case length is again visible as part of the opaque<> data representation). In some cases, the RPC stub may only be able to write a small part of the data, e.g., when RPC calls are shipped over UDP.

Note that, although the kernel guarantees that each call is executed as a whole, such that other concurrent write requests do not interfere with other write requests, the order in which different concurrent write calls are executed cannot be predicted.

If an error has occurred, the number of bytes that were actually written should be returned to inform caller that those bytes were written. A subsequent call to write then returns the corresponding error code.

Errors (negative value) can be returned for the usual reasons. In addition, an error is returned

41. Read_toc always returns up-to-date information, regardless of whether finalize has been called to create an up-to-date TOC. The latter is independent of the runtime information provided by read_toc.

when a write is attempted on a persistent segment.

Note that there exists no truncate primitive on this API. I assume that no value is attached to any specific segment_id, so that when a truncate is required, the equivalent action can be taken to delete the original segment, and create an empty segment with the same type and subtype / description fields as the original one. The named fields are significant to the user of the segment, not segmentID. Actually, it is possible to first create a new segment with the same type, subtype and description as the first one, and after that remove the original one, as AOS distinguishes segments only based on segmentID.

As a final note, AOS_SYS segments may not be written to by AppMW. See the description of the AOS_SYS segment type in sec. 4.2 for information.

```
int delete_seg (cookie_t c, int acid, int seg_id);
```

in: cookie_t c, int acid, int seg_id
out: int [retval]

Delete_seg removes the named segment from the AC, if access is allowed and the AC / segment exists. Delete_seg fails on a persistent segment, causing a negative error value to be returned.

```
make_persistent (cookie_t c, int acid, int seg_id);
```

in: cookie_t c, int acid, int seg_id
out: int [retval]

After creation, segments are transient. After calling make_persistent, mode becomes persistent and the segment nor its information in the TOC may be changed (ever) again. This is the case as otherwise it is not possible to guarantee a persistent's segment's safe arrival with the agent's owner over a multi-hop itinerary. An error (negative returnvalue) is returned if some error occurred (e.g., access denied). Succesful operation returns 0 [note, was 1 in previous version of the API].

```
int finalize_ac(cookie_t c, int acid);
```

in: cookie_t c, int acid
out: int [retval]

This call 'finalizes' the AC. An AC has to be finalized prior to ship_ac. Finalize (re)creates the proper checksums (sha1 hashes) for all segments in their toc_entries. The original TOC in segment 0 is now overwritten with the new one reflecting the current AC's content. Furthermore, as part of finalization, AOS creates a signature using its own private key (placed in segment '1'), and adds its own (self-signed) public key certificate (i.e., its endoint certificate) in segment 2.

After finalize is called, a bit is set in AOS which indicates that the AC is finalized. The finalized bit is inspected as part of the ship_ac call. If the finalized bit is not set, ship_ac will return an error immediately.

Any write on one of the AC's segments after finalize, or any other change to the AC, will cause the finalized bit to be unset (reads are allowed). Internally, AOS stores the AC on disk each time finalize is called. This provides a means for (partial) failure or crash recovery (see also sec 5.2).

Storing the AC on disk is done atomically: finalize causes a new AC to be created and signed. The old AC is not overwritten until the new AC is fully written to disk. In fact, the old AC is first renamed as a backup, and the new AC is renamed to the original AC's name (probably ACID in decimal notation, but that is an implementation decision). Then a sync( ) is done to make sure everything is written to disk. AOS may verify whether the AC is correctly written to disk and remove the old (backup) AC once it is certain that the new AC is safely written. Only when AOS is certain the AC is safely written to disk is the finalize_ac call returned and the old AC removed.

Finalize_ac is an atomic operation: only when the new AC is written safely to disk will finalize_ac return succesfully. During finalize_ac, read or write[42] access to the toc and segments 0-2 is not allowed and write-access to segments in the AC is not permitted (read access is). Until finalize has returned without an error, AppMW may not assume that the new version is safely stored; suppose the AC was only written partially, a crash during finalize may still cause the old AC to be restored instead of the new one after the crash.

Note that this form of crash-persistency is not a complete solution for higher layer middlewares or agents. Particularly, an AOS crash may not coincide with an agent or AppMW crash. As noted elsewhere, AppMW may not immediately realize that AOS has crashed. Only when AppMW next invokes a method on AOS will it get an error signifying that AOS has shut down or has been restarted (see reenable_role( )). This means that an agent (or AppMW) which stored intermediate results in the AC but did not yet finalize is unaware of the crash, and thus of the inconsistency of the AC, unless AppMW uses some form of caching which stores intermediate changes to disk. In some cases, AppMW may have to restart all agents that have used their AC to make sure that they become again 'in sync' with their AC. Possibly, agents will at some point in time, become aware of an AOS crash for other reasons as well, for example because existing connections with other agents are closed. However, some AppMW implementations can provide some transparency by transparently reconnecting to other AppMWs / agents after an AOS crash. Other AppMW behaviour may be that all agents are killed and removed from AppMW as soon as an (AOS) error occurs, leaving it up to the application to solve the issue of how and where to restart agents if required. This note simply illustrates that the AppMW designer should also worry about how to deal with AOS crash recovery.

If the old TOC needs to be maintained (e.g., as part of an audit trail), it needs to be copied from segment 0 by MW *prior* to calling finalize_ac. Only then can it be saved from destruction - once finalize is called, segment 0 contains the newly created TOC, the original TOC is then overwritten and forever lost. During finalize, no operations on the AC are allowed (the AC is locked). The same holds during the ship_ac operation. This has to be enforced by AOS.

5.1.3 Endpoints and communication related calls

```
int create_listen_endpoint (cookie_t c, unsigned int required_index,
char sec_suites[][], struct endp* the_endp);
```

In: cookie_t c, unsigned int required_index, char sec_suites[][],

42. Create_segment, make_persistent, etc. write to the toc

Out: int [retval], struct endp the_endp

Create_listen_endpoint creates an externally accessible listen endpoint on AOS to which other AppMW-level processes can connect. The index that the caller requests relative to AOS is specified using the required_index argument. This index corresponds to the (unsigned int) 'index' field in AOS'es endpoint struct. If a required_index is specified which is already in use, create_listen_endpoint returns an error (ACCESS_DENIED). If AppMW does not care about the index, it specifies required_index 0, in which AOS picks an index.

Create_listen_endpoint returns a descriptor which can be used on subsequent accept calls to accept incomming connection requests to this listen endpoint. It can also be passed to select to inspect the status (pending requests) of the endpoint. Note that endpoint descriptors and connection IDs (see connect( ) and accept( )) should come from the same 'descriptor pool', that is, assigned connectionIDs and endpoint descriptors should not intersect.

The returned endpoint_descriptor may be a local descriptor (relative to the role), or it may be the endpoint's index. Whether endpoint_descriptor has global meaning (as the endpoint's index) or local meaning (similar to a filedescriptor in UNIX) depends on the AOS implementation. AppMW should attach no meaning to the returned descriptor. If AppMW wants to find out its index relative to AOS it can find this out by inspecting the index field in the_endp, which is filled in by AOS when the call returns. A return value > 0 implies successful endpoint creation.

Endpoint descriptors (like connectionIDs and XIDs) are coupled to the role that created it, so that only the creating role can accept connections to it or query the endpoint's status with select() using the descriptor.

AOS should keep lower regions of the index space free for explicit index requests from AppMWs for as long as possible. Particularly, what corresponds to the negative range if the index value were cast to a signed int should preferrably be used for 'internal' ports like send ports in the multiplexing protocol (see muxp.x in Appendix A). Index 0 is not allowed in a listen endpoint struct as created by create_listen_endpoint.

The resulting endpoint's full endpoint struct is written in the_endp by AOS. This struct includes AOS'es ScID, ip(v6), tcp port address and the AppMW endpoint's index.

Note, at AppMW level it is probably good practice to publish the sec_suite strings along with the endp struct (or some composed higher-level endpoint struct that contains the AOS endp), so that a connecting entity knows what sec suite to use during connect[43]. However, AOS provides end-to-end sec_suite matching, so publication of sec_suite strings or trust in the published information is not strictly necessary.

If AOS cannot (e.g., it is already taken) or will not create the requested endpoint for some other reason, a negative errorcode is returned. The content of the struct the_endp is then undefined.

---

43. Note that we assume here that a target AppMW knows the sec_suites supported by its own AOS, and that it publishes the intersection between its own sec_suite and that of the AOS kernel it uses. However, we generally expect an AOS to support all sec_suites listed in appendix A (tls.x), and an AppMW to only specify allowed sec_suites from this list. In addition, it is not fatal when an incorrect (e.g., not intersected) sec_suite is published, as far as AOS is concerned - although connecting may then fail. See also section 5.2.

Sec_suites contains a list of strings indicating security suites that the caller finds acceptable for security of the connection. Examples are listed in tls.x in appendix A. If no sec_suite is passed with create_listen_endpoint, this indicates that the caller does not care, and that all sec_suites supported by this AOS instance may be used. An error code <0 (NO_KNOWN_SECSUITE) is returned when AOS does not support any of the requested security suites.

If the endpoint is established succesfully, the_endp is filled in by AOS to contain the full address assiciation (struct endp) of AOS, including AOS'es ScID (endpointID) and the created index. This endpoint struct can then be registered in a location service or passed to some other party so that clients can make connections to the AppMW. to it.

```
int delete_listen_endpoint (cookie_t c, int endp_descr);
```

In: cookie_t c, int endp_descr
Out: int [retval]

Delete_listen_endpoint deletes the named endpoint from AOS, and closes any pending connections that have arrived at the listen endpoint - if c owns this listen endpoint. Otherwise, a negative errorcode is returned. Existing (accepted / established) connections are not influenced by this call.

[new] Delete_listen_endpoint unblocks all waiting accept() calls on this endpoint. These calls are returned with an error AOS_ERROR_IO when this happens. Subsequent invocations using this endpoint descriptor return UNKNOWN_ENDPDESCR. A just arrived or pending request may still be delivered to accept() (rather than accept() returning AOS_ERROR_IO), depending on timing and AOS implementation. Any accept() calls arriving in the kernel after endpoint deletion get an error UNKNOWN_ENDPDESCR.

Any currently pending requests which are not delivered to any current accept() invocation in the kernel, will receive a TARGET_UNKNOWN error at the connect() side.

```
int connect (cookie_t c, struct endp* target, char sec_suites[][]);
```

In: cookie_t c, char sec_suites[][], struct endp target
Out: int [retval], struct endp target

Target specifies the address of the listen endpoint of a target AppMW process. The scid field of target is set by AOS to indicate the scid of the target. If the scid was already specified (i.e., non-zero) by the caller, the connection only succeeds if the scid corresponds to the self-certifying ID of the AOS that is listening at the ip/port address in the target endpoint (IP/port and index would have to be specified correctly for the connection to succeed, obviously); otherwise, and error AUTHENTICATION is returned[44].

AOS always attempts to authenticate the target AOS. If scid was not specified, it is filled in by the local AOS to match the target's scid, or it is left zero if the target could not be authenticated. If scid was specified in target, the peer must be authenticated using the specified scid, or connect

44. AOS can immediately clean up the connection if authentication did not yield the correct scid, which avoids build-up or keeping of AOS state for this invalid connection.

will return an error AUTHENTICATION.

[new] Note that normally, all AOSes have a ScID and should be able to authenticate themselves. Particularly, authentication is mandatory when using ACTP (the ship_ac / wait_ac calls) succesfully. However, it is conceivable that an AOS is used to only host anonymous services. This may occur when AOS is used for an insecure AppMW-level service that does not want to be bothered with keys, ScIDs, and certificates. In that case, the AOS kernel used by the service may not authenticate itself. If the target could not be authenticated or provided an expired certificate when a zero scid was specified in target, the scid field will be left zero by connect. It is up to the connecting AppMW to determine if a zero ScID in the returned target is acceptable or not[45].

A case where an AOS kernel is also seen as unauthenticated is when it provides an expired or invalid self-signed certificate for authentication[46]. If an expired certificate was provided and a non-zero (matching) scid was specified in target, connect() will also return an error AUTHENTICATION[47].

Retval is a positive connection identifier (conn_id) or a negative error code. **[new]** ConnectionID may not be assumed to have global meaning, it may (or may not) be a local descriptor relative to the calling role. Conn_id is used for subsequent calls like send, recv, select and close.

An error CONN_SETUP is returned if some (transient) error occurred while attempting to connect. This may be caused by a local error (e.g., insufficient resources to set up a new base channel) or by a remote error (e.g., too many pending connections). CONN_SETUP is a transient error, connect may be retried. In some implementations, AOS may itself try to reconnect transparently from the caller before returning an error CONN_SETUP to the calling AppMW. Other errors may be related to secsuite agreement (see below), or because the target listen endpoint is not known to the target AOS (e.g., when the endpoint was deleted). The latter error is indicated by TARGET_UNKNOWN. Also, the peer AOS may be unreachable (PEER_UNREACHABLE). These and other error codes are discussed in errorcodes.h in Appendix B.

The caller may pass a series of cipher suites that it finds acceptable for this connection, in which

---

45. We will attempt to support connections without authentication in AOS v5. If this is not succesful or causes problems in the implementation, please report this to me so we can revise this decision for AOS v6. For overall API testing, we can stick to authenticated kernels only for now, although if you do implement support for zeroed ScIDs, you should test for correctness, of course (if your kernel does *not* or cannot support zeroed ScIDs, please report an AUTHENTICATION error at the API level when an authentication error occurs when scid was zero in the provided target endp struct. Testing kernels without authentication is not a priority as the decision to allow it is sort of tentative.

46. Although note that this is really a different thing and perhaps this should be specifically noted to the calling AppMW by a special error. For v5 we stick to the semantics as described here: both cases yield a zeroed ScID and this is allowed by AOS if a zero scid was specified in the target endp. It is then up to AppMW to determine if an unauthenticated AOS is acceptable.

47. Note that as an alternative for applications that do not care about security, a key pair can be generated with a self-signed certificate with an 'infinite' expiration time, when AOS is configured. Here, AOS has a valid ScID, but the scid field in the (returned) endp struct can be ignored by AppMW. This may be a better solution, as it uses the same mechanism in both secure and insecure cases. We leave a final decision on how to handle unauthenticated AOSes to AOS v6.

case the connection may only use one of the provided cipher suites. Some cipher suites are listed in tls.x in appendix A. If sec_suite of source and target AppMW do not match, an error SECSUITE_AGREEMENT is returned. An error NO_KNOWN_SECSUITE is returned if none of the secsuite strings is supported by this AOS kernel.

If the caller does not pass a cipher suite, the connection may be insecure (non-confidential) or it may use any cipher suite supported by this AOS kernel. The channel *will* be MACed, in general (we currently only allow base channels with message authentication (MACs)). Non-matching sec_suite strings are ignored; connections are negotiated using any recognized sec_suite string. Only if sec_suite contains strings but none of these is recognized by AOS, is an error returned immediately.

```
int accept (cookie_t c, int endp_descr, struct endp* from, bool block);
```

In: cookie_t c, int endp_descr, bool block
Out: int [retval], struct endp from

Accept returns the next incoming connection on a previously created listen endpoint, endp_descr. Endp_descr was returned by create_listen_endpoint. This endpoint descriptor is coupled to the role that created it in AOS: any attempt to call accept using an endpoint descriptor that was not created by this role returns an error UNKNOWN_ENDPDESCR. **[new]** If the used listen endpoint is deleted using the delete_listen_endpoint primitive, accept() is unblocked and returns an AOS_ERROR_IO error. Subsequent accept calls using this endpoint descriptor return UNKNOWN_ENDPDESCR.

Accept returns a positive connectionID if a valid connection is accepted. Similar to an endpoint descriptor, a connectionID is generally a local descriptor and should not be assumed to have any global meaning, i.e., it may or may not correspond to some index or multiplexed connection 'port' internal to AOS. The returned connectionID has a value > 0. An error is returned with a negative return value.

If block is specified, accept waits until a connection is available, otherwise it returns immediately, with a connection or without one. If no connection was pending on the endpoint, 0 is returned. Accept may only be used to wait for connections on a specific endpoint (descriptor). If an AppMW uses, for example, a single thread to accept on multiple endpoints, select can be used to inspect the status of a range of endpoints (if an endpoint has status readable that means that there is a pending connection request, see select( )). An endp_descr with value 0 may not be specified and returns an error INVALID_ARGUMENT.

from is a datastructure that AOS fills in with the endpoint specification of the actual connecting MW. If unauthenticated, the AOS ScID is zeroed.

Note that this call will only return connections that are permitted according to the sec_suites specification of create_listen_endpoint( ). The call will not return when sec_suites do not match, AppMW remains ignorant of connection requests with non-matching sec_suites.

Block specifies if accept() should block or just poll. Accept with block = 1 (TRUE) does not return until a connection request is accepted or the endpoint is deleted (see delete_endpoint). Accept() with block = 0 returns immediately, with 0 if no connection was pending, or with a positive connectionID if one was pending. It may at some time prove useful to replace accept's block parameter with a timeout, but it is very hard to tune this. For that reason, we stick to a boolean block argument.

It is up to AOS to decide how long to leave a connection request to a listen endpoint pending until accept() is called. **[new]** If within this AOS-defined pending time no accept is called, an error CONN_SETUP is returned at the connect side, who then may retry later or give up. Accept() does not notice this error (accept only returns when a connection has succesfully been set up or when the listen endpoint has been deleted). Internally to AOS, an error TOO_MANY_PENDING_CONNECTIONS is returned to the connecting AOS if its pending time is exceeded (see muxp.x in Appendix A). Obviously, the caller (AOS or AppMW) may retry connecting to the endpoint if this error occurred (see also connect()).

```
int send (cookie_t c, int conn_id, int len, char* msgbuf);
```

In: cookie_t c, int conn_id, int len, char msgbuf[]
Out: int [retval]

Send writes len bytes from msgbuf onto the connection. These bytes are sent (and received) in-order with respect to previous or subsequent sends. Send is successful, unless the connection has been closed since the previous send or an I/O, or PEER_ROLE_DELETED (when target role is deleted, and thus this connection is closed) error occurs. UNKNOWN_CONNID is returned when the specified connID is not known or when conn_id is not owned by the invoking role[48].

Send is atomic with respect to the number of bytes written to the channel as indicated by the return value. No bytes from other threads (in AOS) writing on the same connection will be interleaved with the bytes succesfully written on the channel as indicated by the return value. This comment applies to all I/O related methods, such as read/write_seg and recv( ).

[new] Note that no guarantee is given that all requested bytes will be shipped to or from the kernel at once (depending on the RPC stub / dispatcher implementation). Particularly, some RPC implementations will only be able to write a limited amount of bytes to the kernel at once (for example due to restrictions in the code generated by rpcgen for RPC over UDP, see sec. 5.2). Thus, if different AppMW threads share a connection, they have to implement synchronization if they want to avoid that blocks of written data are interleaved.

The number of bytes returned to the caller by a succesful send() call is guaranteed to be sent as a contiguous sequence of bytes and will be received in one piece at the other end of the AppMW-to-AppMW channel (see also recv()).

```
int recv (cookie_t c, int conn_id, int len, char* msgbuf, int block);
```

---

48. **[new]** Note that when a conn_id is owned by another role it may in some cases be more precise to report ACCESS_DENIED (i.e., resource not owned by caller). However, connection IDs may not have global meaning, i.e., they may be local descriptors relative to the calling role (see also connect() and accept()). As the result is irrelevant to the caller anyway, UNKNOWN_CONNID is returned when connID is not known relative to the calling role. A similar issue applies to wait_ac (see wait_ac).

In: cookie_t c, int conn_id, int len
Out: int [retval], char msgbuf

The opposite of send: recv. This call blocks (if block is true) until one or more bytes (up to len) are obtained from the channel; if nonblocking, msgbuf is filled in with the amount of bytes available at that time, which may be 0. If blocking, recv waits until 1 or more bytes are available from the connection, and returns the number of read bytes. The distinction between blocking and non-blocking recv is that a non-blocking recv may return 0 bytes, and a blocking recv call may not (unless an error occurred). The received bytes are placed in msgbuf, and the received (written) number of bytes is indicated by the returnvalue of recv. **[new]** The received bytes (the number returned) are contiguous, as in read from the connection in one blow: recv() is atomic with respect to the number of bytes read. AOS enforces this by avoiding interleaving of recv( ) calls using the same connID (see also send( )).

A negative returnvalue signals an error condition, such as the channel being closed since a previous send or receive, or UNKNOWN_CONNID when the specified connID is not known or not issued to this role.

```
int peek(int connID, int len, char* msgbuf, int block);
```

In: cookie_t c, int conn_id, int len
Out: int [retval], char msgbuf[]

Peek is identical to recv, except that the bytes are not removed from the AOS'es buffer space and can be read again.

```
int close(cookie_t c, int conn_id);
```

In: cookie_t c, conn_id
Out: int [retval]

Close closes the connection identified by connID. Pending unsent data will still be delivered (barring failures) as long as the remote party does not close as well[49]. send( ) and receive( ) calls on the connection are illegal following this call.

Note that modulo that AOS can authenticate the connecting party, when necessary, connections are always established and passed up once accept returns (there is no explicit reject). A close on the connection is required to close (or reject) the connection.

```
int select(cookie_t c, int setsiz, int read_dsc[], int write_dsc[], int
except_dsc[], bool block);
```

In: cookie_t c, int setsiz, int read_dsc[], int write_dsc[], int except_dsc[], bool block
Out: int [retval], int read_dsc[], int write_dsc[], int except_dsc[]

49. This mimics socket close with SOCK_LINGER option. If at some time it turns out to be necessary, a close_immed option to close could be added to specify immediate removal of the socket, including removal of any unsent data buffers.

The select call is analogous to and semantically identical to the socket (BSD UNIX) select call. The syntax and arguments have changed somewhat compared to that call, for reasons explained below. Select is specifically intended for communication related descriptors, that is, it will only work for listen endpoints and connection identifiers (ACs are generally assumed writable / readable, and the ACID space is independent of the connection / endpoint descriptor space, i.e., may intersect).

The interface described here is the C interface for the select call. Other client stub bindings for the interface may have a slightly different interface, for example, the Java representation of this call may omit the setsiz parameter and have objects containing the descriptors instead of arrays of integers as in the C interface. Although this issue exists for all API methods, I denote it here specifically because the representation on the wire for the C library may be slightly different than that for Java; this is allowed and the difference is described below. This description describes the C interface and the XDR on-the-wire representation specifically as an example.

The xxx_dsc arrays are called descriptor sets or arrays and contain sets of endpoint descriptors / connection IDs which are to be monitored on specific conditions[50]. The semantics of the arguments of AOS'es select call are largely equivalent to the arguments to BSD UNIX' socket select. The xxx_dsc are inout parameters: they are used by the caller to specify sets of descriptors, and are also used (by the client stub / library) to place the results of the call in.

read_dsc specifies a set of descriptors which is to be checked on readable status. Particularly, a readable descriptor is a connectionID for which demultiplexed data is ready for a subsequent recv( ) call. A readable endpoint descriptor has one or more connection requests pending.

write_dsc **[new]** is included for versions of AOS which support flow control. Implicitly, all AOSv5 kernels implement flow control, although it is not part of the multiplexing protocol (see muxp.x in Appendix A), but rather an inherited feature from the underlying TCP protocol. Usually, a connectionID is writable, unless the kernel is out of bufferspace to contain the data to be written, or when when the peer stopped reading from the base connection for similar reasons[51].

At the API level, send() may get an error IO_TRANSIENT when a connection is not writeable (or it may block shortly while AOS tries to send the data). **[new]** Although at the API level (send call) this transient error is reported, this particular error is *not* shown as an exception in except_dsc; this 'error' status is only visible because the connection is marked as non-writeable, which is visible by it not being returned in write_dsc by select()[52].

50. Note, the descriptors of connections and endpoints (relative to a role) do not intersect, see create_listen_endpoint.

51. **[new]** AOS is assumed to have (ring)buffers for sending and receiving data, in which it stores outgoing and incoming data for AppMW. These (ring)buffers should be allocated per peer, to avoid one AOS kernel swamping the (incoming) buffers with data, thus denying service to other AOSes. When a peer stops reading from the base connection, for example because its in-buffer (recv buffer) is full, the sending AOS notices that because it cannot write to the TLS/TCP connection, which does implement flow control. At some time, its outgoing (ring)buffer gets full, resulting in an ERROR_IO_TRANSIENT for subsequent send() calls. This sets the status of the virtual connections using that base connection to non-writeable. (Note that in addition, VCs may be allocated a slice relative to the total amount of space in the send (ring)buffer, implementing some kind of QoS or fairness protocol with respect to the different VCs in the kernel. Generally, the amount of space in a (ring)buffer should be roughly proportional to the number of virtual connections using the base connection).

except_dsc specifies a set of descriptors which are to be checked on exceptions. Currently, I/O errors and closed connections are among the conditions that raise an exception status.

If a caller is blocked on select() while waiting for input from a connection and the sending process closes the socket, select() notes this as an exception rather than as data. Therefore, if select() is not currently looking for exceptions, it blocks indefinitely (this corresponds to BSD socket select semantics).

Using the xxx_dsc parameters, the caller specifies a set of connIDs/descriptors which are to be monitored for the status corresponding to the arguments, as described above. In the C interface, NULL may be specified for any set in which the caller is not interested.

A descriptor set is an unordered set of descriptors, which may contain zeros at arbitrary places. 0 descriptors are not interpreted by AOS, only nonzero (positive) descriptors are evaluated by the kernel, zero descriptors are ignored[53]. A descriptor set may never contain more than AOS_MAX_SELECT_SET_SIZE (256) entries.

[new] Currently, a role may invoke at most one (1) select() call at a time. If a role invokes select() while another select() call is in the kernel, the latest invoked select() call will return an error ACCESS_DENIED. Select is usually called to poll the status of a set of descriptors. An AppMW should call select multiple times in a row (polling) with different descriptor sets if it wants to know the status of more than 256 descriptors[54].

If a set contains an unknown positive descriptor, or a descriptor which was not issued to the invoking role, an error UNKNOWN_CONNID is returned, and the corresponding descriptor is placed in the first field of all non-NULL descriptor sets that were passed to select(). No valid descriptors are evaluated when an UNKNOWN_CONNID error is returned.

[new] In the C interface, *setsiz* specifies the size of the non-NULL xxx_dsc sets as the size of the descriptor arrays as the number of descriptors including zero descriptors (so *not* the buffer length in bytes). In the C implementation, this means that any non-NULL descriptor set has to be passed to the select call as a buffer of at least setsiz descriptors. Setsiz may not be larger than AOS_MAX_SELECT_SET_SIZE (256)[55].

Note that, for example, a Java implementation will probably pass objects containing sets of descriptors to the corresponding Java call, which will probably be translated by the Java marshalling routines in a XDR data structure without any zero fields on the wire. However, as noted above, a specific library used by AppMW has the freedom of marshalling descriptor sets as

---

52. I believe this corresponds to BSD socket select semantics. I am not hundred percent sure; please let me know if this is (not) the case.

53. A reason for allowing zero descriptors in the descriptor sets, is to allow an AppMW to easily manage descriptor sets in pre-allocated (fixed) descriptor set arrays, without having to re-order or compact those arrays prior to invoking select().

54. Some future (v6?) AOS version may support concurrent invocations of select with disjunct descriptor sets, similar to wait_ac, with an error code DESCR_IN_USE similar to wait_ac's XID_IN_USE. However, we chose the simpler option of disallowing concurrent select() calls as we think concurrent invocations are probably not required for normal select() use in any AppMW implementation, and also because it can easily be simulated by multiple (polling) subsequent select invocations.

arrays of any size (setsiz) up to 256 bytes, possibly containing zero's, and shipping that to the kernel. The AOS kernel must be able to handle any incoming (possibly sparse) descriptor set of up to 256 entries.

The descriptors from the in parameters are evaluated by the kernel, and the descriptors with the status corresponding to the parameter (as described above) are returned to the caller in the corresponding 'out' argument. The out argument contains an unsorted list containing the descriptors of which the corresponding status was set. The descriptors returned by the kernel are placed at the beginning of the out parameter (without zero's inbetween, in arbitrary order). There are never more descriptors written into the out parameters than there were in in the in argument.

Note that the difference between this call and UNIX socket select is that in the latter bitsets are passed to the operating system, rather than sets of descriptors. The reason is that in UNIX, listen sockets and connections are always identified with a local filedescriptor, and local filedescriptor tables are assumed to be no longer than 1024 entries (in the current BSD sockets ipmlementation). For that reason, a bitset of 1024 bits (one per descriptor that needs to be inspected) is passed in UNIX select.

As we cannot assume anything on the return values of create_listen_endpoint or accept/connect, except that they are in the positive range of a signed integer, specifying a bitmap is unfeasible here. The setsiz argument loosely corresponds to the maxfd argument in BSD socket select, but see above for the exact semantics.

Corresponding to socket select, AOS select returns the sum of all 'ready' descriptors. This means that the sum over all returned descriptors (in the out buffers) is taken. Specifically, analogous to socket select, any descriptor which is returned in more than one descriptor set, is also counted more than once[56]. If no descriptors from any of the in parameters have their corresponding status set, 0 is returned. Negative return values are reserved for the usual errors like IPC_ERROR or UNKNOWN_CONNID, as described above and in errorcodes.h in appendix B.

The block argument has the following semantics: set to 0 (false) means poll the status of the specified descriptors and return immediately, set to 1 (true) means that select may only return if at least one of the specified descriptors has a corresponding status. This means that when block is true, select may only return a value >0 or an error (<0), but not 0.

---

55. This means that in the case of an XDR encoded select argument, an array of setsiz integers can be passed to the AOS kernel (e.g., as an XDR-encoded array of setsiz descriptors) for any non-NULL input descriptor set. Otherwise a zero-length integer array is passed to the kernel instead. The setsiz argument itself is *not* part of the marshalled data shipped to the kernel, in fact, the argument may be omitted for non-C client bindings of the AOS API. An alternative C interface could have contained a length parameter for each descriptor set, but this was not chosen as it would clutter the interface. Note that it is also possible for the C client stub to inspect the descriptor sets and only ship the non-zero entries of the set to AOS. However, this is neither required nor adviced.

56. Why this is done this way in sockets select is unclear to me. However, Stevens' UNIX network programming, second edition, notes the following: "The return value from this function indicates the total number of bits that are ready across all the descriptor sets." "Early releases of SVR4 had a bug in their implementation of select: if the same bit was on in multiple sets [...] it was counted only once. Current releases fix this bug." As not counting descriptors this way is considered a bug in UNIX select, it is probably a good idea to use the UNIX select call's way of computing the return value. It probably has something to do with being able to stop searching through the returned descriptor sets when [n] descriptors were found.

```
int get_parm(cookie_t c, char parameter_name[]);
```

in: cookie_t c, char parameter_name[]
out: int [retval]

get_parm takes returns the value of an AOS parameter known by name parameter_name. In general, AOS parameters are limits, such as "MAX_ACS_PER_AOS", "MAX_ACS_PER_ROLE" or "MAX_SEGMENT_SIZE". Currently, no parameter names are defined, but these should be described in a different AOS document (possibly for a given implementation). See params.h in Appendix A for a tentative attempt to start defining some parameters.

[new] Returnvalue are a positive value or 0 (the parameter's value) or an error such as PARM_UNKNOWN when the parameter_name is unknown to AOS or this role. A special value PARM_INFINITE is defined to indicate that this parameter is infinite, often implying no limit.

Note that some parameters may be role-specific. For example, if we manage per-role quota's using parameters, these are role-relative. In addition, some parameters (e.g., DISP_MAX_RPC_MSG_LENGTH, see sec. 5.2) may be queryable by any role including a zero cookie.

For now, the meaning and usage of this call and its parameters are still experimental. We intend to finalize the call's semantics and a number of parameters for v6. In v5, the method may be used but if so it should be noted that nothing about its usage is currently final (see e.g., sec. 5.2 for some issues).

```
int reenable_role (cookie_t c);
```

in: cookie_t c
out: int [retval]

reenable_role is used to reset the role so that it can be used again. After a crash (or graceful exit where roles are not deleted) of AOS, all roles of AOS are locked and method invocations by a role return AOS_RESET so that the role becomes aware that AOS has crashed, which means that some of the role's resources are lost, such as communication endpoints and open connections. A role can reenable itself by calling reenable_role. After reenable_role is called, the calling role can again use AOS.

An error UNKNOWN_ROLE is returned if cookie is unknown.

5.2 A Note on RPC Message Sizes [new in rc3]

Some RPC variants, most notably SunRPC over UDP, pose constraints on the maximum message size (of the marshalled message). For example, SunRPC over UDP compiled by Linux rpcgen has a maximum message size of 8091 bytes. Dependent on the implementation, either the AOS dispatcher or the client AppMW stub may impose such a message size limit.

To handle this issue, the client stub can quench its read / write requests (e.g., recv()) to a number of bytes that it can handle. Similarly, the dispatcher can impose a limit on read or write requests.

If the dispatcher imposes a limit, it is imperative that the client stub knows the limit, as otherwise it may send a larger message to the kernel than the dispatcher can handle (e.g., when writing or sending a chunk of data). The dispatcher exports a parameter DISP_MAX_RPC_MSG_SIZE (using the get_parm() API method) which defines its maximum message size, if applicable.

Quenching read / write requests to support RPC message size limits

Some methods of the AOS API have fixed maximum sizes for their arguments which are below the minimal RPC message length of 8091 bytes. Other methods have variable size arguments, which have to be modified to the stub, resp. dispatcher's maximum message size.

All affected methods are described in this section. Methods which are not mentioned have arguments which fit easily within the minimal RPC message size of 8091 bytes. The AOS_MAX_... parameters described below are defined in params.h and can be queried using get_parm(). I remove the AOS_MAX_ prefix from the errors named below for readability.

prepare_wait_ac: the sec_suites argument is restricted to size SEC_SUITE_SIZE (see params.h). This value is currently fixed to 4096 bytes, which is way enough to contain any set of currently known security suite strings.

read_toc: the n parameter is sensitive to the maximum message size. The client stub should quench the request to an n whose return (tocent list) will fit in its own maximum message size. In addition, the dispatcher should adapt n so that its own limit is not exceeded, before calling the AOS native interface. The kernel will then not return more tocents than the dispatcher can ship back to the stub. Optionally, the stub can already limit the requested n to the dispatcher's maximum message size but this is not mandatory as the dispatcher already implements this limit.

Read_toc's semantics allow returning less than n tocents, even if more are available (see **[new]** description of read_toc).

ship_ac: sec_suites argument is limited by SEC_SUITE_SIZE parameter.

create_seg: subtype and description string sizes have a maximum size of SUBTYPE_SIZE and DESCRIPTION_SIZE (see params.h).

read_seg: The n argument is sensitive, however, the call's semantics allow returning a number of read bytes less than the requested size, even if more data is available. The stub limits n to a size fitting in its maximum message size. The dispatcher imposes its own limit before calling the native method of the AOS kernel, so that no more than the minimum of the dispatcher's and the stub's max message sizes are returned by the kernel.

write_seg: the stub applies the minimum of its own limit and DISP_MAX_RPC_MSG_LENGTH to determine the number of bytes that it ships to the dispatcher, so that the overall message length does not exceed either stub or dispatcher's maximum message length.

create_listen_endp: the sec_suite size is limited by the parameter SEC_SUITE_SIZE.

connect: the sec_suite size is limited by the parameter SEC_SUITE_SIZE.

send: the len argument is sensitive, however, the call's semantics allow for sending a number of bytes less than the specified len. The stub limits len to the minimum of its own maximum message size and DISP_MAX_RPC_MSG_LENGTH, so that the overall message length does not exceed either the stub or the dispatcher's maximum message length.

recv and peek: The len argument is sensitive, however, the call's semantics allow returning a number bytes less than the requested amount, even if more data is available. The stub limits len to a size fitting its own maximum message size. The dispatcher in addition adapts len to its own maximum before calling the native method of the AOS kernel, so that no more than the minimum of the dispatcher and the stub's max message sizes are returned by the kernel.

select: the read_dsc, write_desc and except_desc sizes are limited by the SELECT_SET_SIZE parameter.

get_parm: the parameter_name size is limited by the PARM_STRING_SIZE parameter.

Obtaining maximum RPC message size information.

The client stub's limit is implicit, and implemented in the client stub. The dispatcher's limit has to be made public to the client stubs by the dispatcher. This can be handled in two ways:

- The message size is noted in the file containing the dispatcher's endpoint and the AppMW cookie, which is read by AppMW when it starts up.
- The message size can be obtained by querying the parameter DISP_MAX_RPC_MSG_LENGTH using the get_parm() API method. The get_parm method and parameter string size (max. 128 characters) is small enough that it will work with any dispatcher. This parameter is returned by the dispatcher, so any dispatcher can set its own DISP_MAX_RPC_MSG_LENGTH parameter.

Of these methods, the latter is always usable. The first method is optional; a particular AppMW implementation may or may not use message size related information available from its startup file. If an AOS kernel publishes maximum message size information for its dispatchers in the AppMW startup file, it has to make sure this matches the value of the DISP_MAX_RPC_MSG_LENGTH parameter (see params.h), as a stub may then not use the get_parm() method and can assume the value from the startup file to be correct.

Note that an AppMW may be implemented as multiple processes implemented in potentially different languages. These processes may each use a different dispatcher. A stub needs to be initialized with the transport endpoint that it has to use, and be fed with the parameter at initialization time by AppMW, or alternatively AppMW can call get_parm first thing and call a private method on the stub to initialize it with the message length[57]. As it queries the parameter directly from its own dispatcher, it can be sure that the value is correct for that dispatcher.

57. It is also possible to let the stub query the dispatcher itself, but that requires initializing the stub with a cookie using which it can call on the dispatcher. Or we can allow get_parm(DISP_MAX_RPC_MSG_LENGTH) to be called with a zero cookie (see get_parm() description - some parameters may be dependent on or protected by role information. This may open up the possibility of allowing queries for non-role owners for certain parameters like DISP_MAX_RPC_MSG_LENGTH, i.e., for these parameters we may not check cookie. Any role may then invoke get_parm, checking is done in the method itself. Todo see if this is feasible.).

The get_parm() method is mandatory and can always be used, i.e., the dispatcher always has to set the DISP_MAX_RPC_MSG_LENGTH parameter. However, see the footnote above - let's settle this issue for now some way or another (i.e., pick one), and postpone a definite decision until AOSAPIv6.

For now let's say: allow get_parm() for DISP_MAX_RPC_MSG_LENGTH to be called with zeroed cookie, and support that in the kernel. If you think it is more convenient, place this information (also) in the AppMW startup file, and let us (me) know how you did it. We define the kernel to support the DISP_MAX_RPC_MSG_LENGTH parameter with zeroed cookie, whether your AppMW uses this or another mechanism is up to you for now[58].

5.3 On Security Suite Negotiation over a Shared Base Channel

Ship_ac, connect, prepare_wait_ac and create_listen_endpoint all take their own cipher suite specifications, which may be any subset of the cipher suites that AOS supports. For that reason, for any 'connection' between a pair of AppMW level endpoints to succeed, the cipher suite matching has to take place at a higher level than TLS. However, one particular problem with this is that at the server side, the specific cipher suites of the virtual endpoints are hidden behind the tcp listen port that the client AOS first has to connect to. The result may be that a connect to a tcp endpoint (i.e., matching client AppMW's sec_suite with target AOS'es sec_suite) succeeds, but incorrectly so as the negotiated cipher may not match with the cipher suite specified by the target endpoint.

A solution is to create a new tcp listen endpoint for each server-side AppMW listen endpoint. In itself, from the API viewpoint, there is nothing wrong with this, the new tcp endpoint will be returned to the caller and connections can be made to this tcp endpoint to reach the AppMW's endpoint. However, when we want to be able to deploy AOS behind, for example, a firewall this will be problematic as such dynamically allocated tcp listen ports will generally not be reachable through a firewall or NAT router.

Instead, a target ('server side') AOS establishes the union of the sec_suites of all AppMW-level listen endpoints or the complete set of all AOS-supported secsuites, and attaches that to its tcp (TLS) listen endpoint[59]. A client AOS sets up a new connection to the target AOS to negotiate an active suite using the client AppMW's sec suite, unless a base connection to the target AOS already exists with an active suite that is also in the client AppMW's sec suite. If a base connection can be established using one of the client AppMW's sec suites, this base connection is used to send the client AppMW's communication connect request (or ship_ac request) to the target AOS.

58. Implementation should be easy, if AOS hardwires a role with zero cookie that only allows invocation of the get_parm() method. This role can query all non-role specific information available using get_parm(), which seems OK to me. Note that any role-specific information can only be filtered if the native method of the get_parm() call gets the cookie as an argument and uses it to check whether specific information may be queried, or uses it to look up particular information from the per-role tables maintained in the kernel.

59. The set of all AOS-supported secsuites can straightforwardly be enabled for AOS'es tcp listen endpoint. However, using the union of the AppMW endpoints instead already cuts off connection requests at the base AOS-to-AOS level when a client AppMW sec_suite is used that is completely disjunct with any of the target AppMW-level listen endpoints. Simply enabling all AOS supported secsuites and always doing endpoint secsuite matching at the AppMW-level endpoints after demultiplexing is probably simplest from an implementation perspective.

At the target side, when the request comes in, AOS checks whether the active suite of the base channel over which the request came matches any of the target endpoint's security suites. If this is the case, the connection request may proceed. If this is not the case, the MUXP or ACTP request (internal to AOS) returns an error to the client AOS indicating a secsuite intersection mismatch at the target AppMW level endpoint. The error also returns the intersection between the target AppMW's secsuite and the target AOS'es supported secsuites (of the TCP endpoint) to the connecting AOS which an be used by the source AOS to transparently (to the caller AppMW) reconnect with the target AOS'es tcp endpoint to set up a base connection that is guaranteed not to cause a secsuite intersection error with the target AppMW's listen endpoint[60]. To achieve this, the client AOS intersects the returned secsuite with the client AppMW's security suite, and uses this intersection to select another existing base channel which intersects with the resulting intersection, or to set up a new base channel using this intersection.

If the connecting AOS does not have sufficient resources to set up a new base channel (e.g., when currently out of filedescriptors), the client AOS returns an error CONN_SETUP to the calling AppMW. **[new]** Removed some stuff related to session management which is obsolete in view of the base connection management protocol (BCMP) which is to be introduced in AOS in AOSAPIv6.

Note that base channels should probably use some kind of reference count mechanism to establish whether they are still in use by any ACTP or MUXP protocols. Unused base channels should be released, to save resources (i.e., filedescriptors)[61].

5.4 A Note on Kernel Shutdown and Crash Recovery

When a kernel shuts down or crashes, roles and agent containers which were finalized are retained on disk. Completely received and verified ACs as part of ACTP (even if they are not collected yet by wait_ac) are considered finalized and kept, including their XIDs so that they can be collected by a subsequent wait_ac call using that XID. ACs and XIDs of unfinished ACTPs are deleted resp. invalidated.

Connection endpoints and communication channels are transient and are not kept when the kernel shuts down or crashes. Endpoints and connections should be recreated by AppMW after kernel restart (and reenable_role, see reenable_role() call).

A 'hard' (kill -9) exit or a crash of the kernel should not cause role information to be deleted, unless it was marked for deletion. This has to be ensured by writing all 'persistent' role state to disk (and call sync( )) timely. After a kernel has been restarted, roles that invoke a method will get an error ERROR_AOS_RESET from the kernel, to indicate that the kernel has been restarted,

60. Note that once negotiated active suites of a given base channel may never be renegotiated, since otherwise the resulting active suite may fault other current users of the base channel. The target AOS should make sure that no client AOS is able to do a renegotiation to a different security suite (and take care that any such an event does not close the base connection) - although note that a compliant client AOS should never attempt a secsuite renegotiation (TLS rehandshake) over the base channel, so if it does so after all it is its own due when the base connection is closed as a result. A workaround is probably possible, however this probably also introduces sufficient complexity that I consider it better to forbid TLS (sec suite) rehandshaking of the base channel.

61. Saving on resources such as open tcp connections was one of the reasons for using multiplexing in the first place.

so that AppMW can take measures to recover (e.g., reconnect to AppMWs, create new listen endpoints). In the meanwhile, it is possible that AppMW already obtained an ERROR_IPC. In order to be able to invoke roles normally again, the role has to call reenable_role( ) on the restarted kernel.

It will be troublesome to keep track of processes which hold roles as they are not started up by AOS itself. Due to this, it may not be possible to notice that a process holding a role has crashed, and that the corresponding role should be deleted. We have to assume that AppMW keeps track of its child role-keeping processes and cleans up roles (some time after) a child process has died, unless it has reason to keep the role alive, e.g., because grandchildren are still doing important stuff.

Unless deleted, a role's information and associated state and resources (e.g., ACs, listen ports, pending connections, etc.) are kept for a reasonable time nearing 'infinite', say a week after the role was last used. An 'epoch' count may be used to note the number of crashes that a role underwent, which is reset (to 0) when reenable_role is called by AppMW. This prevents large state-buildup after a number of successive kernel restarts; after a number of restarts during which a role was not used or reenabled, the role may be assumed to be out-of-use and can be cleaned up. Note that in this case a reasonable time should have been available between kernel restarts, so that roles had the chance to reenable themselves. So, an additional 'survival time' (much less than a week, obviously) has to be accounted for in the garbage-collection algorithm to address this aspect. A role should only be deleted when all of its descendants have also expired, to avoid aberrant behaviour[62].

Note, all ACTP requests that were not finished are cancelled, and all allocated XIDs are released. Only ACs that were stored on disk and whose corresponding create_ac/wait_ac/finalize_ac calls were really finished (i.e., returned) are kept after a kernel restart. Unfinished wait_ac requests are cancelled and the possibly received ACs for ACTPs in progress are deleted. Note that for this reason, the receipt sent by the receiving AOS after AC verification / acceptance by the kernel should be shipped to the sending AOS only *after* wait_ac returned.

To remove all old AOS (role-related) state from the kernel, the kernel must be explicitly told to do so when started up, using a special argument like 'aos -c' (for 'clean start'). This tells the kernel to clean up all its state.

[new] Note that in addition, or actually preferrably, init can / should be used to delete all AppMW roles (top-level roles) and their role subtrees explicitly, rather than using aos -c.

Aos -c is usable as a fallback mechanism. Aos -c should basically implement delete_role (init_role's cookie), i.e., it also deletes the init role, so that a fresh init role can be created and init can be started afresh if the old init process somehow got into an inconsistent or erroneous state.

Additional information about crash/restart persistence can be found in the description of the

62. Particularly, if this constraint is not implemented, child roles may be deleted before their time and/or may not be deletable in the normal way any longer because their parent no longer exists. This depends on implementation, in particular whether the child roles are also deleted or only the expired roles themselves. To avoid such implementation or design decisions, the constraint is given here that an expired role may only be deleted if all its (grand)children have also expired.

delete_role and finalize_ac calls.

6. Appendix A: Protocol specification for AOS v5. AOS-level internal protocols.

This section contains the basic internal protocol specifications that AOS uses for multiplexing connections and the agent container transfer protocol (ACTP).

Current described protocol: AOS protocols version 1.0.0 (version applies to all protocols).

The specifications are placed in separate files, each for a specific XDR protocol. The following files exist:

```
proto.x: overall protocol header and miscellanious protocol-related
         definitions.
toc.x:   The specification of TOC related data structures.
muxp.x:   The  protocol  headers  (in  xdr)  needed  for  the  multiplexing
protocol.
actp.x:  The protocol headers for the Agent Container Transfer Protocol
(ACTP).
tls.x:   A header containing definitions for TLS related error handling.
ap.x:    A template definition for the Agent Passport.
params.h: A tentative specification of some parameters for get_parm()
```

```
/*============================== proto.x ================================*/

/* Note: uses MUXP_MSG and ACTP_MSG definitions. Defined in muxp.x and
   actp.x, respectively. No need for #include. rpcgen can live without them
   and jrpcgen can't live with them ;o) */


/************************ General Message Layout ************************/

/*
  All kernel protocols always have a standard header in front of each message,
  (the same header in both directions) to distinguish different protocol
  messages from each other, i.e., MUXP from ACTP messages, if shipped over
  the same wire. PROTO_MSG is also used without interleaving of protocol
  messages.

  Note, PROTO_MSG contains a length field which contains the precomputed
  length of the whole message following PROTO_MSG, including header.
  So, body_length is size of ACTP_MSG or MUXP_MSG. This may help getting the
  message off the wire as soon as possible, without having to decode the
  XDR-encoded body first.

  When adding protocols, add its discriminator to the enum PROTO_ID and
  increment PROTO_VERSION.
 */

enum PROTO_ID {
  MUXP,
  ACTP
};
```

```
/*
  Overall protocol version. Independent of sub protocol version (MUXP /
  ACTP version), though a different PROTO_VERSION may mandate that the sub
  protocols version changes as well (i.e., PROTO_VERSION supercedes
  subprotocol version numbers.)
 */
const PROTO_VERSION=1000000; /* i.e. 1.0.0 */

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

struct PROTO_MSG {
  unsigned int proto_version; /* always PROTO_VERSION. */
  unsigned int body_length;   /* precomputed total length of body. */
  PROTO_BODY body;            /* the message body */
};

/*============================ toc.x ======================================*/

/*
  Defines several AOS kernel data structures related to the TOC (table of
  contents) of an agent container:

    * The format of a TOC entry (TOC_ENTRY) as presented to the middleware
      when the middleware queries the TOC.

    * The format of the AOS kernel's TOC entry and TOC data structures as
      written by the agent container finalization code. These are called
      FINALIZED_TOC_ENTRY and FINALIZED_TOC.


   About the finalized TOC:
   The TOC, signature and certificate files of the signer are placed in the
   AC zip-file as files 0, 1 and 2, respectively. Segments 0, 1 and 2 *ARE*
   listed in the TOC, but with zeroed size and checksum fields, as these
   would cause a self-reference (it is impossible to include checksums for
   segment 0 and 1 as the moment that they are written in the toc, the toc and
   the signature changes).

   The TOC does list the segments 0-1 because the metadata about those segments,
   particularly the description and subtype fields, may be useful. In here it
   may, for example, be noted which signature algorithm is used or what format
   the key (in segment 2) is in. Specifically what fields are set in the TOC:
   SegID is set, segtype is set, mode is set to TRANSIENT,
   creation_date may optionally be set (prior to signing, of course),
   last_modification_date is 0, seg_size is 0, and checksum is 0. Subtype and
   description may be set if required.

   Note that segments 0-2 are visible including valid size fields to an AppMW at
   the receiving AOS using read_toc: their size fields are generated by the AOS
   kernel based on the actual segments 0, 1 and 2 which it obtained from the
   actual AC zip file. The toc obtained by reading segment 0, however, always
   contains zeroed checksum and size fields for segment 0-2, as segment 0
   contains the raw, 'on-the-wire' toc.
```

```
   */

/********************************** Misc **********************************/


/* Null-terminated ASCII string. */
const SUBTYPE_SIZE = 256;
typedef opaque SUBTYPE [SUBTYPE_SIZE];

/* Null-terminated ASCII string. */
const DESCRIPTION_SIZE = 256;
typedef opaque DESCRIPTION [DESCRIPTION_SIZE];

/* SHA-1 checksum. */
const SHA1_SIZE = 20;
typedef opaque DIGEST[SHA1_SIZE];
typedef DIGEST CHECKSUM;

typedef int SEGMENT_ID;
/*
  Segment ID for finalization segment containing the serialised TOC (see
  FINALIZED_TOC).
*/
const SEGMENT_ID_TOC = 0;

/*
  Segment ID for finalization segment containing the signature over the
  serialized TOC. This is an SHA-1/RSA signature in base-64 encoding.
*/
const SEGMENT_ID_SIGNATURE = 1;

/*
  Segment ID for finalization segment containing the key certificate of an AOS
  kernel. The key described by the certificate can be used to verify the
  signature over the serialised TOC. This is an X509 certificate in PEM
  encoding.
*/
const SEGMENT_ID_CERTIFICATE = 2;


/***************************** Segment Types *****************************/

enum SEGMENT_TYPE {
  SEGMENT_TYPE_ZERO,    /* for deleted segs */
  /* Type of segment that contains a serialised TOC */
  SEGMENT_TYPE_TOC,
  /* Type of segment that contains signature over a serialised TOC. */
  SEGMENT_TYPE_SIGNATURE,
  /* Type of segment that contains the key certificate of an AOS kernel. */
  SEGMENT_TYPE_CERTIFICATE,
  /* Type of segment that contains code to be executed by the middleware. */
  SEGMENT_TYPE_CODE,
  /* Type of segment that contains data defined by the middleware / agent */
  SEGMENT_TYPE_DATA,
  SEGMENT_TYPE_AGT_PASSPORT,
  SEGMENT_TYPE_AOS_SYS,
```

```
  SEGMENT_TYPE_APP_SYS
};


/****************************** Segment Modes ******************************/


enum SEGMENT_MODE {

  /* Segment may be modified and removed. */
  SEGMENT_MODE_TRANSIENT,

  /* Segment may not be modified or removed. */
  SEGMENT_MODE_PERSISTENT
};

/*
  Seconds since 1-1-1970
 */
typedef int TIME;


/******************************** TOC entry ********************************/


/*
  TOC entry as presented to the middleware by a call to read_toc(). Defined in
  XDR for implementation convenience only.
*/

struct TOC_ENTRY {

  /*
     Identifies the segment within the agent container.
   */
  SEGMENT_ID segment_id;

  /*
     Non-negative number that identifies the type of segment.
   */
  SEGMENT_TYPE segment_type;

  /*
     Describes whether the segment is mutable.
   */
  SEGMENT_MODE segment_mode;

  /*
     Time at which the segment was created (the number of seconds since
     January 1, 1970, 00:00:00 GMT).
   */
  TIME creation_date;

  /*
     Time at which segment content was last modified (the number of seconds
     since January 1, 1970, 00:00:00 GMT).
```

```
   */
  TIME last_modified_date;

  /*
     Size of the segment data measured in bytes.
   */
  unsigned int segment_size;

  /*
     Subtype attached to the segment by the user, for example AppMW can
     place a code subtype string for a code segment here.
   */
  SUBTYPE subtype;

  /*
     Description attached to the segment by the middleware. Not interpreted by
     the AOS kernel. An example use is for the middleware to name the data
     segments of an agent container and to place the name of a segment in this
     field.
   */
  DESCRIPTION description;
};


/****************** Finalized ('on the wire') TOC entry *******************/

/*
  TOC entry as formatted by the agent container finalization code. See
  TOC_ENTRY documentation for the meaning of common fields.
*/

struct FINALIZED_TOC_ENTRY {

  SEGMENT_ID segment_id;

  SEGMENT_TYPE segment_type;
  SEGMENT_MODE segment_mode;

  TIME creation_date;
  TIME last_modified_date;

  unsigned int segment_size;

  SUBTYPE subtype;
  DESCRIPTION description;

  /*
     SHA-1 checksum over segment content.
   */
  CHECKSUM checksum;
};


/*********************************** TOC ***********************************/

/* Current version of the TOC format. */
```

```
const TOC_VERSION = 1000000; /* i.e. 1.0.0 */

/*
  TOC as formatted by the agent container finalization code.
*/
struct FINALIZED_TOC {
  /*
     Version number of the TOC format used. Must be TOC_VERSION.
   */
  unsigned int toc_version;

  /*
     Variable-length array of TOC entries. TOC entries are ordered by their
     segment ID. Note: TOC may contain empty slots for removed transient
     segments (see discussion in AOS design note document). An empty slot is
     completely zeroed.
   */
  FINALIZED_TOC_ENTRY entries<>;
};

/*============================== tls.x ==================================*/

/*
  Some definitions for the kernel's protocols related to Transport Layer
  Security [RFC 2246].
 */

/*
  A security suite / cipher suite spec as defined in RFC2246, Appendix C.
  String is not zero terminated.
  */
typedef string  TLS_SEC_SUITE<>;

/* Set of suites */
typedef TLS_SEC_SUITE TLS_SEC_SUITES<>;

/* For your information, the strings considered useful in AOS at the
   moment, are listed below. These do include SSL_* strings, but note that
   these are not officially registered like TLS_* suites in the TLS RFC, and
   mostly correspond to TLS strings. Also, usage of these stings may
   imply usage of the SSL handshake and key-exchange algorithms, rather
   than the (public) TLS algorithms. Usage of the TLS_* cipher suites
   is therefor preferred, although most TLS implementations also support SSL
   ciphers, and thus AOS can also use them. We include the SSL strings for
   completeness.
   Note that *anon* strings from RFC2246 are not supported, as we want
   mutual authentication using RSA.
   Strings including DSS are not listed as we currently use only
   RSA keys (any size). DSS may be supported in the future, though.
   Also note that export key sizes (40 bits) are supported, although
   these ciphers are considered very weak and use should be avoided.
    (I believe export restrictions no longer apply to the ciphers listed
     below - but please check this before you use it.)
   As a further note, I omit TLS_NULL_WITH_NULL_NULL, as this is not
   a legal cipher suite and is only used in TLS during the initial
   handshake phase. Use TLS_RSA_WITH_NULL_MD5 or TLS_RSA_WITH_NULL_SHA
```

```
   if you want no encryption (no confidentiality). Of the two,
   TLS_(SSL_)RSA_WITH_NULL_MD5 is assumed to be fastest.
 */

/*
 At the moment the following cipher suites are proposed to be
 supported by all AOS kernels:

    ** --> Not from the TLS v1.0 RFC. May be useful but  **
    ** not sure.                                         **

    SSL_RSA_WITH_RC4_128_MD5
    SSL_RSA_WITH_RC4_128_SHA
    SSL_RSA_WITH_3DES_EDE_CBC_SHA
    SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
    SSL_RSA_WITH_DES_CBC_SHA
    SSL_DHE_RSA_WITH_DES_CBC_SHA
    SSL_RSA_EXPORT_WITH_RC4_40_MD5
    SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
    SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
    SSL_RSA_WITH_NULL_MD5
    SSL_RSA_WITH_NULL_SHA
    TLS_RSA_WITH_AES_128_CBC_SHA --> Note: not in TLS v1.0 spec!
    TLS_DHE_RSA_WITH_AES_128_CBC_SHA --> Note: not in TLS v1.0 spec!

    ** --> From the TLS specification v1.0 (RFC 2246).     **
    **     We should probably stick to the ciphers listed  **
    **     below. Note that the export ciphers should      **
    **     preferrably not be used (I believe export       **
    **     restrictions do no longer apply).               **

    ** Export ciphers. These are considered                **
    ** very weak and should be avoided if possible.        **

    TLS_RSA_EXPORT_WITH_RC4_40_MD5
    TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
    TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
    TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA

    ** These cipher suites definitely apply.               **

    TLS_RSA_WITH_NULL_MD5
    TLS_RSA_WITH_NULL_SHA
    TLS_RSA_WITH_RC4_128_MD5
    TLS_RSA_WITH_RC4_128_SHA
    TLS_RSA_WITH_IDEA_CBC_SHA
    TLS_RSA_WITH_DES_CBC_SHA
    TLS_RSA_WITH_3DES_EDE_CBC_SHA

    ** I'm actually not sure about DH(E)_RSA but I think   **
    ** these are also possible.                            **

    TLS_DH_RSA_WITH_DES_CBC_SHA
    TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
    TLS_DHE_RSA_WITH_DES_CBC_SHA
    TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
```

```
 */


/*============================== muxp.x ===================================*/

/* Note: uses TLS_SEC_SUITES definition from tls.x. No need for #include.
   rpcgen can live without them and jrpcgen can't live with them ;o) */

/*
  Wire protocol specification for the AOS kernel's multiplexing protocol.
  This specification is complemented by the finite state machine of the
  protocol.

  Connections
  -----------

    The multiplexer protocol implements 'virtual connections' on top of a
    lower-level connection-oriented 'base' protocol (such as TCP/IP).
    Connections provided by the base protocol ('base connections') are
    assumed
    to be reliable and ordered, and virtual connections of the multiplexer
    protocol inherit these ordering and reliability properties.

    A virtual connection is bound to a single base connection, meaning that
    messages pertaining to the virtual connection are carried by that base
    connection alone.

  Port numbers
  ------------

    First note that although multiplexer port numbers are defined somewhat
    analogous to TCP port numbers, they are independent of TCP port numbers.

    There are two kinds of multiplexer port numbers: listen port numbers and
    connection port numbers. Both are defined to be positive integers.

    Listen port numbers are used to establish virtual connections.  A listen
    port number is effectively an address within a process at which virtual
    connections may be accepted. Typically a listen port number is combined
    with an address defined by the base protocol (e.g. an IP address plus
    TCP listen port) to form a complete address.

    Connection port numbers are used to identify an established virtual
    connection.  A virtual connection is assigned a pair of connection port
    numbers, each of which is provided by a process that participates in the
    connection.  Each process independently controls the allocation of its
    connection port number space, subject to the following restriction:

      * The connection port number that a process assigns to a virtual
        connection must uniquely identify the virtual connection relative to
        the combination of (a) the process, and (b) the base connection to
        which the virtual connection is bound.

    Note that due to this restriction a single connection port number is
    already be sufficient to identify a virtual connection. We use a pair of
    connection port numbers merely for implementation convenience and
```

```
    robustness.

  Flow control
  ------------

    This version of the protocol does not address flow control of virtual
    connections.

  Denial of Service
  -----------------

    The MUXP protocol cannot really do anything against DoS attacks.
    Specifically, MUXP does not place any constraints on how often
    virtual connections are used; there may be channels alive for
    weeks at an end without any apparent use (e.g., to wait for the result
    of some simulation). This is an AppMW design issue.

    The most that can be done (and probably should be done) by AOS
    is detection of setup of a large number of base connections
    or virtual connections (MUXP_OPEN messages) without apparent use at
    the same time.
    Then it is probably legitimate for AOS to close the (base) connection(s)
    from that AOS, as a DoS attack may be taking place.
    Of course, if AOS crashes or is restarted, all (virtual) connections
    are automatically released.
*/

/* version of the multiplexer protocol */
const MUXP_VERSION = 1000000; /* i.e. 1.0.0 */

/* maximum message length in this version is 32k */
const MUXP_MAX_MSG_LENGTH = 32768;

/*
  Connection or listen port number type. A valid port number is a positive
  integer. We use a signed type for implementation convenience.
*/
typedef int MUXP_PORT_NUMBER;


/*
  Only used in OPEN_NACK messages
 */
enum MUXP_OPEN_ERROR_CODE {

  /* no-one was listening, standard error */
  MUXP_OPEN_ERROR_PORT_UNKNOWN,

  /* This error is returned if the peer requires a separate base connection
   * for this MUXP protocol.
   * Particularly, this error may be returned when MUXP_OPEN_MSG was sent
   * over a base connection already in use for ACTP, and the target AOS
   * implementation wants to keep MUXP and ACTP on different connections.
   * The error says it all: select another, or create a new, base
   * connection and try again.
   */
```

```
  MUXP_OPEN_ERROR_USE_SEPARATE_BASECONN,

  /* This error is returned if create_listen_endpoint invoked at the peer
   * has a sec_suite which does not intersect with the active sec suite on
   * the current base connection.
   * A new, fresh base connection has to be set up using the a sec suite from
   * the intersection of the sending and the target sec suites. The
   * target sec suite is returned in the MUXP_OPEN_NACK_MSG when this
   * error (NACK) occurs.
   * Alternatively (preferrably), any base connection that has an active sec
   * suite from the intersection of sending and target sec suites may be
   * used.
   */
  MUXP_OPEN_ERROR_BASECONN_SECSUITE_INTERSECT,

  /*
   * This error is returned in MUXP_OPEN_NACK_MSG when there are too many
   * pending virtual connections for the targeted endpoint at the peer.
   * The number of pending connections is not settable in the AOS API.
   * It is up to AOS to choose a reasonable limit, depending on its
   * implementation. A retry may be attempted, obviously.
   *
   * Currently, this error is also returned if the peer decided that it
   * kept the incoming connection pending long enough (i.e., accept() was
   * not called often enough). See description of accept() in the API
   * document.
   */
  MUXP_OPEN_ERROR_TOO_MANY_PENDING_CONNECTIONS
};


/*
  Only used in CLOSE messages
 */
enum MUXP_CLOSE_ERROR_CODE {

  /*
   * Used in a MUXP_CLOSE_MSG to signal that a virtual connection was
   * closed gracefully and not because of role deletion.
   *
   * This 'error' occurs as the result of a close() of the VC by AppMW.
   *
   * Note that endpoint deletion does not influence existing connections,
   * as described with the call delete_listen_endpoint. In that case,
   * existing VCs are not closed, so no special error code (PORT_DELETED)
   * is required for this.
   */
  MUXP_CLOSE_ERROR_NO_ERROR,

  /*
   * This error allows for internal consistency checks:
   * a MUXP message using some unknown port may be shipped by an AOS
   * kernel that was erroneous, malicious, or otherwise did not do
   * a MUXP_OPEN first.
   */
  MUXP_CLOSE_ERROR_PORT_UNKNOWN,
```

```
  /*
   * This error is returned if the peer closed the virtual connection
   * because the owning role was deleted via the delete_role kernel
   * primitive. Note that delivery of pending messages over the
   * closed VC is not guaranteed when a role is deleted,
   * as would be the case with a normal AOS API close() call.
   * See description of the delete_role call in the API.
   */
  MUXP_CLOSE_ERROR_ROLE_DELETED,
};




/* the different messages */
enum MUXP_MSG_TYPE {
  MUXP_OPEN, MUXP_CLOSE, MUXP_OPEN_ACK, MUXP_OPEN_NACK, MUXP_PAYLOAD
};


/**************************** Specific Messages ****************************/

/*
  GENERAL NOTE: for each message description, references to processes are in
  terms of the 'sender' and 'receiver' of that particular message. For example,
  the 'sender_cport' field in a message is the connection port assigned by the
  process that sends the message.
*/

/*
  MUXP_OPEN_MSG

  A request to open a connection port in the receiver for the purpose of
  establishing a virtual connection. Carries the listen port number of the
  receiver and the proposed sender's connection port number for the new
  connection.
*/

struct MUXP_OPEN_MSG {
  MUXP_PORT_NUMBER sender_cport;
  MUXP_PORT_NUMBER receiver_lport;
};

/*
  MUXP_OPEN_ACK_MSG

  A positive response to MUXP_OPEN_MSG. Carries the connection port numbers of
  the sender and the receiver of this message for the new virtual connection.
  The receiver connection port number is the same as the sender port number in
  the corresponding MUXP_OPEN_MSG.
*/

struct MUXP_OPEN_ACK_MSG {
  MUXP_PORT_NUMBER sender_cport;
  MUXP_PORT_NUMBER receiver_cport;
```

```
};

/*
  MUXP_OPEN_NACK_MSG

  A negative response to MUXP_OPEN_MSG. Carries the listen port number of the
  sender and the connection port number of the receiver of this message,
  plus error information. These match the port numbers that were used in the
  corresponding MUXP_OPEN_MSG.
*/

union MUXP_OPEN_ERROR switch (MUXP_OPEN_ERROR_CODE code) {
  case MUXP_OPEN_ERROR_BASECONN_SECSUITE_INTERSECT: TLS_SEC_SUITES receiver_sec_suites;
  default: void;
};

struct MUXP_OPEN_NACK_MSG {
  MUXP_PORT_NUMBER sender_lport;
  MUXP_PORT_NUMBER receiver_cport;
  MUXP_OPEN_ERROR error;
};


/*
  MUXP_CLOSE_MSG

  To fully close a virtual connection, both participating processes must
  send a MUXP_CLOSE_MSG for the virtual connection.
  After sending MUXP_CLOSE_MSG a process must not send further messages
  pertaining to the virtual connection,
  but must be prepared to accept further messages pertaining to the
  virtual connection until it has received a MUXP_CLOSE_MSG.

  Connection port numbers remain assigned to a virtual connection until both
  sides have sent MUXP_CLOSE_MSG for the virtual connection.
*/

struct MUXP_CLOSE_MSG {
  MUXP_PORT_NUMBER sender_cport;
  MUXP_PORT_NUMBER receiver_cport;
  MUXP_CLOSE_ERROR_CODE error_code;
};

/*
  MUXP_PAYLOAD_MSG

  Encapsulates payload transmitted by a higher layer through the virtual
  connection.

  Note: Although no explicit upper bound on the size of the payload is
  specified here, an implicit upper bound exists due to the maximum size of a
  message.
*/

struct MUXP_PAYLOAD_MSG {
  MUXP_PORT_NUMBER sender_cport;
```

```
  MUXP_PORT_NUMBER receiver_cport;
  opaque payload<>;
};


/*************** GENERAL MUXING PROTOCOL MSG LAYOUT ********************/

union MUXP_MSG_BODY switch(MUXP_MSG_TYPE type) {
  case MUXP_OPEN      : MUXP_OPEN_MSG      open_msg;
  case MUXP_OPEN_ACK  : MUXP_OPEN_ACK_MSG  open_ack_msg;
  case MUXP_OPEN_NACK : MUXP_OPEN_NACK_MSG open_nack_msg;
  case MUXP_CLOSE     : MUXP_CLOSE_MSG     close_msg;
  case MUXP_PAYLOAD   : MUXP_PAYLOAD_MSG   payload_msg;
};

struct MUXP_MSG {
  unsigned int muxp_version; /* always MUXP_VERSION */
  MUXP_MSG_BODY body;        /* the message body */
};

/*================================ actp.x ===================================*/

/* Note: uses TLS_SEC_SUITES definition from tls.x. No need for #include.
   rpcgen can live without them and jrpcgen can't live with them ;o) */

/*
  Agent Container Transfer Protocol (ACTP). This is the protocol used by AOS
  kernels to ship agent containers.

  ACTP messages as described in this file may in theory be carried by
        1. a MUXP virtual connection.
        2. a base connection (e.g. TCP) together with MUXP
        3. an unshared base connection.
  In all cases, the actp messages are embedded in a PROTO_MSG (see proto.x).

  In AOS API v5, ACTP over MUXP is forbidden. In v5, prepare_wait_ac
  may only return an endpoint with index 0, i.e., using a nonmultiplexed
  base connection to AOS. Because from a protocol construction viewpoint
  there is no objection to shipping ACTP messages over a MUXP connection,
  it is discussed here also for completeness.

  When ACTP uses its own unshared base connection to AOS this allows
  for its own set of security protocols (sec_suites). This prevents
  interference and conflicts between (security) properties set for the
  MUXP and the ACTP protocols.
  This base connection is possibly established using the same listen-port
  as the MUXP protocol.

  Each ACTP transaction ships a single agent container and is identified by
  a transaction ID (xid), which is an 128 bit random number.  The
  transaction ID is established outside the scope of this protocol.

  Given that AOS kernel S wishes to ship an agent container to AOS kernel R,
  the message exchange of a successful transaction between these two AOS
  kernels looks as follows. Note that this description covers the most
  general case, where ACTP can also be shipped over a virtual (multiplexed)
```

```
  channel. This scenario is forbidden in AOS v5.

    * S kernel opens a (virtual) connection (v)c to R.

    * S sends an ACTP_SHIP_MSG to R using the (v)c. Note, all messages of the
      ACTP contain the xid field.

    * After R receives ACTP_SHIP_MSG, it sends ACTP_SHIP_REPLY_MSG to S
      using the (v)c. The ACTP_SHIP_REPLY_MSG contains a status code and,
      if RESULT_OK, the TOC signature and R's key in an ACTP_RECEIPT struct.

    * After R has sent ACTP_SHIP_REPLY_MSG it closes the (v)c.

    * After S has received ACTP_SHIP_REPLY_MSG, it closes the (v)c.

  In the case of an unsuccessful transaction, S and/or R may abort the
  transaction by closing the (v)c before completing all of the above steps.
  However, if R wishes to abort, it should if possible send a SHIP_REPLY_MSG
  containing an ACTP_ERROR struct before closing the (v)c.

  ACTP_SHIP_MSG may contain fragments of an AC, rather than the whole
  AC in one go. An  ACTP_SHIP_REPLY_MSG is sent only once for each incoming
  AC, so not for each fragment.
  Particularly, an error (e.g., XID_UNKNOWN) may be sent at any time
  during AC (fragment) shipping. It is up to the shipping side to check
  (or peek) for incoming ship_reply messages some time during or
  after shipping AC fragments.
  The same holds for ACTP_ERROR_USE_SEPARATE_BASECONN.
  The receiving side should be prepared to receive all fragments of an
  AC, even if it returned one of the above errors, as it does not know
  when the reply will be read by the sending AOS. Of course it can
  throw away fragments that it will not use (if an error is returned).
  Also, the base connection may immediately be closed after an error
  was returned, if possible (i.e., if no other ACTPs or MUXPs use the
  same base connection).

  CONSTRAINTS:
  To protect against certain attacks, XIDs should be used within a certain
  timeframe. After this time, XID is automatically de-allocated by AOS.
  The timeframe is an AOS-defined parameter, probably an hour or so,
  which can be obtained using AOS'es get_parm() call.
  ACTP_INITIAL_XID_INVALIDATION_TIME is the (tentative) name of this
  parameter (see params.h).

  Note that the xid invalidation timeframe applies only to the *first*
  ACTP message sent.
  After that, XID is bound to a given client AOS (through the authenticated
  channel to which the ACTP is bound). This means it is secured against
  attacks from outsiders.
  An ACTP may, after its initial message containing XID,
  take as long as it wants to ship the actual AC (i.e., it may use a very
  slow link to ship the AC over).
  Clearly, other errors may occur during this (slow) shipment, such as
  an error that the shipped AC is too long for the target AOS. This
  is described above and later in this file.
*/
```

```
/*
  Transaction ID of an agent container shipment.  The transaction ID is a
  128 bit number in big-endian order.
 */
const ACTP_XID_BYTELENGTH=16;
typedef opaque ACTP_XID[ACTP_XID_BYTELENGTH];

/*
  Result code describing the outcome of an ACTP transaction.
 */
enum ACTP_RESULT_CODE {

  /* Successful shipment of an agent container. */
  RESULT_OK,

  /* Remainder indicate usuccessful shipment of an agent container. */
  RESULT_NOT_OK
};


enum ACTP_ERROR_CODE {

  /* This error is returned if the XID is already in use, i.e., the
   * ACTP has already been initiated by some other AOS (or another
   * ship_ac thread in the same AOS). Note that there is no risk of
   * misusing this information if another AOS is using XID, as
   * the receiving AOS will only allow ACTP messages with this XID
   * from the original initiating AOS over the encrypted, authenticated
   * AOS-to-AOS channel originally set up between the sending and
   * receiving AOS.
   *
   * This is mainly a consistency error, resulting when a bug in
   * a target AOS issued the same XID twice, but it may also occur if
   * some attacker guessed XID and attempts or succeeds to 'hijack' an
   * ACTP and deny legitimate access using XID.
   */
  ACTP_ERROR_XID_IN_USE,

  /* This error is returned if the peer requires a separate base connection
   * for this ACTP protocol.
   * Particularly, this error may be returned when an ACTP message was sent
   * over a base connection already in use for MUXP, and the target AOS
   * implementation wants to keep MUXP and ACTP on different connections.
   * It may also be returned if the peer wants dedicated base connections
   * for every ACTP run. The error says it all: select another, or create a
   * new, base connection and try again.
   *
   * XID is not invalidated with this error.
   */
  ACTP_ERROR_USE_SEPARATE_BASECONN,

  /* This error is returned if the wait_ac invoked at the peer has a
   * sec_suite which does not intersect with the active sec suite on the
   * current base connection.
   * A new, fresh base connection has to be set up using the a sec suite from
   * the intersection of the sending and the target sec suites.
```

```
 * The target sec suite is returned in the ACTP_ERROR message when this
 * error occurs. Alternatively (preferrably), another base connection that
 * has an active sec suite from the intersection of sending and target
 * sec suites may be used to retry the ACTP.
 *
 * XID is not invalidated with this error.
 */
ACTP_ERROR_BASECONN_SECSUITE_INTERSECT,

/* The XID is unknown to this AOS kernel. */
ACTP_ERROR_XID_UNKNOWN,

/* The agent container is too large to store at AOS. This is a
 * hard, fixed limit set by the peer. When AOS is currently out of
 * disk, for example because it already has too many AC's accepted,
 * ACTP_ERROR_PEER_TRANSIENT is returned instead.
 *
 * XID is invalidated when this error occurs.
 */
ACTP_ERROR_AC_TOO_LARGE,

/* The agent container could not be unmarshalled.
 *
 * XID is invalidated when this error occurs.
 */
ACTP_ERROR_AC_FORMAT,

/* The agent container could not be verified.
 *
 * XID is invalidated when this error occurs.
 */
ACTP_ERROR_AC_VERIFY,

/* A transient error, like temporarily out of diskspace, occurred.
 * A retry can be attempted in general (e.g., to see if agents moved
 * in the meanwhile). ACTP_ERROR_PEER_TRANSIENT typically indicates resource
 * shortage (peer 'busy').
 * Note that the sending AOS may retry shipment transparently, without
 * ship_ac returning in the meanwhile, or it may return an
 * error AOS_ERROR_ACTP_PEER_TRANSIENT to ship_ac, so that AppMW may
 * retry.
 *
 * XID is not invalidated when this error is returned.
 */
ACTP_ERROR_PEER_TRANSIENT,

/* An internal error occured at the peer. This is generally a
 * persistent error, indicating some 'fatal' error condition like a
 * disk crash or something similar.
 *
 * XID is invalidated when this error occurs.
 * Note: the peer will _try_ to invalidate XID, but whether this
 * succeeds depends on the internal problem(s) that it faces..
 */
ACTP_ERROR_PEER_INTERNAL,
```

```
  /*
   * This error is returned when the peer aborts the transaction because
   * the owning role was deleted via the delete_role kernel primitive.
   * This error is only returned when the role (and thus XID) is deleted
   * during an existing ACTP session. This error is returned only once.
   * Subsequent ACTP messages for this XID, and other cases where XID is
   * (no longer) known will receive an ACTP_ERROR_XID_UNKNOWN instead.
   *
   * XID (obviously) is invalidated when this error occurs.
   */
  ACTP_ERROR_ROLE_DELETED
};


/* version of the protocol */
const ACTP_VERSION = 1000000; /* i.e. 1.0.0 */

/* the different messages */
enum ACTP_MSG_TYPE {
  ACTP_SHIP,
  ACTP_SHIP_REPLY
};


/****************************** Specific Messages *****************************/

/*
  ACTP_SHIP_MSG

  This message ships an agent container. It carries the transaction ID for
  this shipment and the marshalled agent container. The agent container is
  a zipfile. The zipfile consists of a set of files, one file for each of
  the AC's segments. The name of a segment file is the (integer) segment
  ID of the corresponding segment in decimal notation, without leading
  zeros. (Note that in itself, the toc + segments form a kind of protected
  zip/tar-file. However, placing the AC in a zip file allows us to
  transport it opaquely as a whole to another AOS which can handle the
  zip-file offline without interpreting the toc on the fly to extract the
  segments).
 */

struct ACTP_SHIP_MSG {
  ACTP_XID xid;
  bool more_fragments_follow;
  /* if more_fragments_follow = 0, end of AC is reached. */
  opaque agent_container<>;
};

/* Note: fragment max. size is not yet determined for this ACTP version.
 * It is up to the sender to determine size of (optional) fragments.
 * An AC may be shipped at once, but for large AC's this uses
 * up all the bandwidth, which is considered a bit greedy when base
 * connection is potentially shared with other ACTP or MUXP messages. */

/*
  ACTP_SHIP_REPLY_MSG
```

```
  This message is sent in response to ACTP_SHIP_MSG and contains the
  outcome of the shipment.
*/


union ACTP_EXTENDED_ERROR switch (ACTP_ERROR_CODE code) {
  case ACTP_ERROR_BASECONN_SECSUITE_INTERSECT:
       TLS_SEC_SUITES receiver_sec_suites;
  default: void;
};


struct ACTP_ERROR {
  ACTP_EXTENDED_ERROR ext;
  string optional_verbose_debug_string<>;
};


struct ACTP_RECEIPT {
  opaque toc_signature<>;
  opaque toc_cert<>;
};


union ACTP_RESULT switch (ACTP_RESULT_CODE code) {
  case RESULT_OK: ACTP_RECEIPT receipt;
  case RESULT_NOT_OK: ACTP_ERROR error;
};


struct ACTP_SHIP_REPLY_MSG {
  ACTP_XID xid;
  ACTP_RESULT result;
};


/************************** General Message Layout **************************/

union ACTP_MSG_BODY switch(ACTP_MSG_TYPE type) {
  case ACTP_SHIP        : ACTP_SHIP_MSG  ship_msg;
  case ACTP_SHIP_REPLY  : ACTP_SHIP_REPLY_MSG ship_reply_msg;
};

struct ACTP_MSG {
  unsigned int ACTP_version; /* always ACTP_VERSION */
  ACTP_MSG_BODY body;        /* the message body */
};

/*================================ ap.x =================================*/

/* This file contains an XDR specification for the Agent Passport (AP).
 *
 * It is intended for use in any AppMW system, although an AppMW is
 * free to change the AP representation. This AP contains fields that
 * allow for application-specific extensions.
 * The AP specification is intended to be general enough to be used in
```

```
 * any mobile agent system using AOS or the AC.
 *
 * Author: Guido van 't Noordende, November 2004.
 */

const AP_VERSION=1; /* i.e., 0.0.1 */

/* Introduction.
 * An agent passport is a set of signatures of an agent's code.
 * Each code segment should (preferrably) be signed by the code
 * author. Each code (or library) segment may have a different author,
 * or multiple segments may have the same author.
 *
 * Furthermore, the AC is signed by the agent's owner.
 *
 * Each AP has a creation and an expiration date, given in UNIX time,
 * i.e., in seconds since Januari first, 1970.
 */

/*
 * The code certificate contains the agent code author's part of the
 * agent passport: basically, the agent code author's part of the AP
 * consists of the hash(es) of the code segment(s) that the author wrote,
 * an expiration date, some additional information, and the author's
 * public key certificate, signed using the private key corresponding
 * to the author's public key stored in the certificate.
 */

struct CODE_CERTIFICATE {
  /*
   * validity info
   */
  TIME start_date;
  TIME expiration_date;

  /* code_checksum matches a checksum in toc entry of code segment.
   * This is simply a sha-1 hash of the code, it can
   * be created offline by hashing the file containing
   * the code. This will match the segment's sha-1 hash in the AC's toc.
   * Note that not all checksums in the code certificate need be
   * present in the AC; however, the code certificate is added to the agent
   * passport if one or more code segments signed by this certificate
   * are present in the AC.
   *
   * Each author may sign any number of code segments in a single
   * certificate. An author may also create a different CODE_CERTIFICATE
   * for every code segment.
   */
  CHECKSUM code_checksum<>;

  /* This code certificate vouches for the authenticity of the code (who
   * wrote it), not necessarily for things like the quality or trustworthiness
   * of the code.
   *
   * about_certificate is defined such that the author's name and some
   * additional information about the author or the code can be stored in
```

```
   * the code certificate as human-readable and/or machine readable text.
   * What kind of meaning is attached to the content of this field is outside
   * the scope of this definition. The field may also contain a legal
   * document such as a disclaimer, and/or a licence file, for example.
   * The field is opaque and may contain any kind or set of documents or
   * any type of data or file(s).
   */
  opaque about_certificate<>;

  /* Author_cert ontains the public key certificate (possibly certificate
   * chain) of the signer in standard x509v3 PEM-encoded format. Placing
   * the signer_cert here effectively makes code_sig self-signed.
   *
   * Key management issues (e.g., trusted third parties vouching for the
   * certificate's authenticity, etc.) are not considered here but are an
   * application issue (there may be a relation to the about_certificate
   * field).
   */
  opaque author_cert<>;

  /* signature's hash algorithm and public key type, standard string.
   * For example SHA-1_WITH_RSA.
   */
  string signature_type<>;

  /* Signature contains the signature of the code by the author,
   * in PEM format (this is standard output of many signature
   * algorithms).
   */
  opaque signature<>;
};

/****** The total agent passport is defined as: *******/

struct AP {
 /*
  * Always equal to AP_VERSION
  */
 unsigned int ap_version;

 /* validity info
  */
 TIME start_date;
 TIME expiration_date;

 /* Context contains a human-readable string describing the context
  * for which the AP is intended. For example, this may contain the
  * name of the AppMW system or a specific Mansion world.
  * Outside context, the AP is invalid.
  */
 string context<>;

 CODE_CERTIFICATE author_code_sigs<>; /* a set of code signatures */

 /* Space for middleware designer to place extra information in or
  * extend the certificate with additional fields.
```

```
  */
 opaque ap_extension<>;

 /* Signer_cert ontains the public key certificate (possibly certificate
  * chain) of the agent's owner in standard x509v3 PEM-encoded format
  */
 opaque signer_cert<>;

 /* Agent owner signature's hash algorithm and public key type.
  * For example SHA-1_WITH_RSA.
  */
 string signature_type<>;

 /* Agent owner's signature contains the signature over all fields of
  * struct AP up to now in PEM format using the algorithm indicated above.
  */
 opaque signature<>;

 /* Applications may require several extensions to the basic AP.
  * For example,Mansion requires that on world entrance
  * the World Entrance Daemon signs the AP as a voucher that the
  * agent is allowed in the world. This signature and related information
  * can be placed in the ap_external_extensions<> field.
  * In this example, in most worlds, another signature by the agent's owner
  * is also required to vouch that the agent owner agreed with the WED's
  * signature (as a WED authentication).
  * Both signatures and possible additional information can be placed in
  * the ap_external_extensions field.
  *
  * Ap_external_extensions is external to the 'basic' AP in that it
  * contains information and/or signatures over all the fields up to now,
  * which constitute the actual agent passport. Defining
  * ap_external_extensions here allows these extensions to be part of the
  * same file / segment in an AC.
  */
 opaque ap_external_extensions<>;

};
/* =========================== params.h ============================*/

/*
 * Initial prototype parameter specification. Subject to change.
 * Date: 22-12-2004, author: Guido van 't Noordende.
 */

/* Current spec is totally C-specific. Parameter names should
 * be identical in any version / implementation (as strings).
 * Parameter strings should have a maximum size of 128 ASCII
 * characters, and should not be NULL terminated (in case of
 * XDR encoding in the SunRPC dispatcher).
 */

/* This file contains some parameters as used by the kernel.
 * Its values are suggestions, and may be overridden to any value
 * by a specific kernel implementation. The parameters can be
 * obtained using the call get_parm() of the kernel API, using
```

```
 * the names below.
 * Note that some values are actually set by the dispatcher,
 * not by the kernel proper. These values are clearly marked.
 *
 * The current file contains conservative values that will be least
 * likely to interference with testing procedures.
 * Changing the values should preferrably be postponed until later in
 * the testing procedures, i.e., until we know more about AppMW behavior
 * in practice.
 */


/*
 * === Some global variables used later in this header file ===
 */

/* This value specifies infinite, e.g., for the AC pending time.
 * We should probably hardcode the MAX_INT value here instead,
 * to avoid that AppMW and AOS assume different values for MAX_INT. TODO.
 * Note we explicitly do not use 0 for this value, 0 may have meaning
 * for some parameters.
 *
 * Note that this value should *not* be interpreted as the actual value:
 * it is special and means 'any value (infinite) allowed'.
 */
#define PARM_INFINITE = MAX_INT; /* TODO hardwire value */

/*
 * === Fixed  limits as defined by the AOS v5 API ===
 */

/*
 * Fixed limits. Listed here only informatively for completeness.
 * The values can be queried if required.
 * Note that querying is not really necessary as the values are fixed
 * for the v5 kernel.
 *
 * -- NEVER MODIFY THESE VALUES !!! --
 */
AOS_MAX_SELECT_SET_SIZE = 256; /* Select's connid sets have max. 256 fields */
AOS_MAX_WAIT_AC_XID_LIST_SIZE = 256; /* max 256 XIDs per wait_ac */
AOS_MAX_SEC_SUITE_SIZE = 4096; /* product size of sec suite list in bytes */
AOS_MAX_PARM_STRING_SIZE = 128; /* In C / XDR: without null termination */
AOS_MAX_SUBTYPE_SIZE = 256;     /* was defined as SSZ in the spec */
AOS_MAX_DESCRIPTION_SIZE = 256; /* was defined as SSZ in the spec */
AOS_SAFE_RPC_MSG_LENGTH = 8091;  /* SunRPC over UDP, minimal size.
                                  * This should _always_ work as a
                                  * minimal value. */

/*
 * This undocumented parameter is currently not used and not described in the
 * v5rc3 API document. I leave it here as it shows that this parameter may
 * be tuned in future versions, however, we currently ignore this parameter
 * in the implementation (i.e., assume it is a fixed 'parameter').
 * --> PLEASE IGNORE THIS PARAMETER.
 * The reason for not using it is that an AppMW may also collect (wait_ac)
```

```
 * or create ACs and not use or delete them. Also, when the AppMW role
 * is delted, the (stale) AC's are removed as well.
 * As we have to assume a well-behaved AppMW that actually uses the
 * XIDs that it creates, we will generally not need this parameter.
 *
 * We keep the parameter in this file so we can (re)open a discussion on it
 * if it appears useful to clean up uncollected ACs sometime in the future
 * (v6)
 *
 * The 'pending time' between full arrival of an AC and its collection
 * by wait_ac. After ACTP_WAIT_AC_PENDING_TIME has exceeded, the AC
 * is removed and the XID becomes invalid (see AOS_ERROR_ACTP_TIMEOUT
 * in errorcodes.h).
 *
 * Note that normally, the owning AppMW should call wait_ac and delete_ac
 * to remove it explicitly after collecting it.
 * In addition note that the AC and XID will also be removed automatically
 * when the owning role is deleted.
 * For this reason, the proposed value is PARM_INFINITE.
 */
AOS_ACTP_AC_PENDING_TIME = PARM_INFINITE; /* Seconds from full AC receipt */


/*
 * === Parameters set by dispatcher ===
 */

/* Maximum size of RPC messages shipped to the kernel, in bytes.
 *
 * We do not specify maxima for the individual fields. The client
 * stub should be 'intelligent', and calculate what the individual
 * argument field restrictions should be. For example, for a write call,
 * the maximum is implemented by restricting the number of written
 * bytes to MAX_RPC_MSG_LENGTH - (sizeof fixed fields in RPC struct).
 * Thus, some parts of the client (and server) stub should be
 * handcrafted to intelligently apply the dispatcher's maximum msg size.
 *
 * Similarly, if the client stub has a (small) max message size,
 * this is handled by the stub by replacing the 'n' or 'length'
 * fields of read calls by a value <= its own max message size.
 * The client stub's max message size is not explicitly negotiated
 * between client stub and dispatcher, it is handled solely by the
 * client stub in the above described way.
 *
 * Note that this only applies to methods which return the number
 * of written (or read) bytes. Certain call which do not return this
 * number to the caller already have fixed limits (e.g., the select
 * set size and the wait_ac XID lists are fixed to a maximum of
 * 256 entries), see the API spec for details.
 *
 * Note that, in general, the stub should call get_parm (MAX_RPC_MSG_LENGTH)
 * when it first comes into action, prior to doing anything else, to obtain
 * the maximum RPC message size limit, and initialize itself accordingly.
 * Get_parm() is represented by a short message, which should always work.
 *
 * As this document is still preliminary, it may still be debated if this
```

```
 * limit should not be in the file read by AppMW prior to initialization,
 * which contains also the cookie and local IPC port used to communicate with
 * AOS. I propose that get_parm() with this parameter will always work,
 * irrespective of whether the size is already somehow contained in the
 * cookie/IPC port file used to initialize AppMW. [Note Arno and Reinier:
 * please sync on the file layout.]
 */

/* Using AOS_SAFE_RPC_MSG_LENGTH is suboptimal if stub and dispatcher
 * allow larger message sizes, as this requires more rpc messages to
 * obtain or send larger chunks of data from the kernel. */
DISP_MAX_RPC_MSG_LENGTH = AOS_SAFE_RPC_MSG_LENGTH;

/*
 * Note, the client stub should adapt any read request to its own
 * maximum before shipping the request (e.g., read of n bytes) to
 * the AOS dispatcher.
 *
 * The same maximum applies to write request: min_msg_size(stub,dispatcher)
 * is used to determine the number of requested bytes from, and the maximum
 * of shipped bytes to, the dispatcher.
 */


/*
 * === AOS Kernel defined parameters - varies between implementations === *
 */

/*
 * The maximum size of an AC (limit may be hit when writing to an AC or when
 * shipping the AC).
 *
 * Actually, it would be preferrable to hardwire this for all vX kernels
 * to ensure interoperability.
 *
 * Note, an AC may also be seen as too large if an AppMW's disk quota
 * ran out (TODO possibly introduce AppMW quota parameters).
 */
AOS_AC_MAX_SIZE = PARM_INFINITE; /* Size in bytes. */

/*
 * The timeframe within which the first ACTP message for a given XID should
 * arrive. After this timeframe, XID is automatically deallocated by AOS,
 * if no ACTP message for this XID arrived. Incoming ACTP requests
 * get the reply ACTP_ERROR_XID_UNKNOWN.
 *
 * The value is in seconds, from the time that 'prepare_wait_ac' is called.
 */
AOS_ACTP_XID_INVALIDATION_TIME = 3600; /* An hour for starters */
```

7. Appendix B: Error Codes of the AOS API v5 rc1 calls

7.1 Error Codes

```
/*========================= errorcodes.h ===============================*/

/*
 * AOSAPI v5 ERROR CODES accompanying the AOS API v5 design notes
 * and the internal AOS protocol specifications.
 *
 * Authors: Guido van 't Noordende and Arno Bakker
 * Date: Oct 29 2004 - Jan 2, 2005.
 *
 * Error codes visible to users of the AOS API (i.e., AppMW))
 * as returnvalues of the API calls.
 * These error codes may be mapped to exceptions in the Java API
 * implementation. This is language specific, but the errors below
 * must be present with the same semantics in any implementation.
 *
 * Internal errorcodes used in the protocols between AOS processes are
 * defined elsewhere.
 */

enum AOS_ERROR_CODE {

    /*
     * 1. Special errors. There is no default error in this group.
     */

    /* For completeness, return value 0 means no error occured. */
    AOS_ERROR_NO_ERROR = 0,

    /* Error during (inter-process) communication between AppMW and kernel.
     * There are no IPC semantics: the call may or may not have been executed
     * by the kernel. */
    AOS_ERROR_IPC = -1,

    /* This error is returned (by any call of a role) when AOS has crashed
     * or was restarted. When that happens, all 'transient' resources
     * (like communication and listen endpoints) of a role are released,
     * and AppMW has to invoke the reenable_role() call in order to resume
     * kernel usage. See the design notes for details.
     */
    AOS_ERROR_AOS_RESET = -2,

    /*
     * 2. Local argument related errors. If it is unclear which error to
     *    return, return AOS_ERROR_INVALID_ARGUMENT.
     */

    /* When an argument passed to a kernel primitive is not valid */
    AOS_ERROR_INVALID_ARGUMENT = -3,
```

```
/* When a cookie (role) is not known by the kernel */
AOS_ERROR_UNKNOWN_ROLE = -4,

/* When attempting to delete an unknown role */
AOS_ERROR_UNKNOWN_ROLEID = -5,

/* When attempting to use an unknown agent container */
AOS_ERROR_UNKNOWN_ACID = -6,

/* When attempting to use an unknown segment */
AOS_ERROR_UNKNOWN_SEGMENTID = -7,

/* When attempting to send/receive from an unknown connection */
AOS_ERROR_UNKNOWN_CONNID = -8,

/* When attempting wait_ac with an invalid XID relative to the calling
 * role (XID may be a valid ('known') relative to another role, this is
 * hidden from the caller of wait_ac).
 *
 * Also returned by wait_ac when a XID has been deleted because ship_ac
 * was not succesfully initiated within XID_INVALIDATION_TIME (params.h).
 * See also AOS_ERROR_TARGET_UNKNOWN in this file.
 */
AOS_ERROR_UNKNOWN_XID = -9,

/* This error is returned when a XID is part of wait_ac's XID list
 * which is already used in another wait_ac invocation.
 * Also returned if ship_ac is invoked with a XID that is already
 * in use at the peer by another ACTP. (Note that locally, ship_ac
 * calls may take the same XID for different actp endpoints,
 * as different AOSes may issue the same XID independently of
 * each other. XIDs are only unique relative to the creating AOS.)
 */
AOS_ERROR_XID_IN_USE = -10,

/* When attempting an accept with an unknown endpoint descriptor */
AOS_ERROR_UNKNOWN_ENDPDESCR = -11,

/* When calling get_parm with an unknown parameter string */
AOS_ERROR_UNKNOWN_PARM = -12,

/* When none of the requested security suites is supported by
 * this kernel*/
AOS_ERROR_NO_KNOWN_SECSUITE = -13,

/* When attempting to write beyond a segment's end */
AOS_ERROR_SEGMENT_HOLE_EXCEPTION = -14,

/* When attempting to write to a segment that has been made persistent */
AOS_ERROR_PERSISTENT_SEGMENT_ERROR = -15,

/*
 * 3. Disk I/O, network I/O and communication errors.
 */

/* Error occured during I/O */
```

```
    AOS_ERROR_IO = -16,

    /* For transient errors occurring locally in AOS. Specifically
     * intended for cases like temporarily out of memory.
     *
     * When a transient io error is returned, the call can be safely
     * retried.
     */
    AOS_ERROR_IO_TRANSIENT = -17,

   /* When (local) AOS has insufficient resources to set up a base
    * connection as a result of secsuite mismatch (requiring a new base
    * channel) or because the peer AOS requires a (separate) base connection
    * for another reason - and the local AOS cannot currently comply.
    *
    * This error also occurs when the maximum of pending connections
    * was reached at the peer, or when the connection pending timeout
    * was exceeded at the peer (peer AOS'es discretion, see description
    * of accept()'s block parameter in the AOSAPIv5 document). In both
    * cases, an error TOO_MANY_PENDING_CONNECTIONS is raised in
    * muxp.x (note that AOS may do some retries first before returing
    * the error to the caller (connect), depending on implementation).
    *
    * Note that his generally a transient error; the operation can be
    * safely retried.
    */
    AOS_ERROR_CONN_SETUP = -18,

    /* Peer AOS could not be physically reached over the network
     *  (i.e. this error does not cover authentication errors)
     * Note: PEER_UNREACHABLE is returned at (base) connection setup time
     * during connect() or ship_ac().
     * When an existing connection breaks (not gracefully), return an
     * AOS_ERROR_IO instead.
     * PEER_UNREACHABLE happens specifically when the tcp connect()
     * times out, indicating a network problem.
     *
     * This error may well be transient, but whether it is handled as
     * transient (retry) or not is up to AppMW. It may take some
     * time before this error is returned, AOS may internally retry
     * a few times before returing the error code, depending on
     * implementation.
     */
    AOS_ERROR_PEER_UNREACHABLE = -19,

    /* The peer AOS does not know the given endpoint index, or when
     * shipping an AC to a kernel that is not waiting for the given
     * transaction ID.
     * Also returned by ship_ac when XID was deleted because ship_ac
     * was not succesfully initiated within XID_INVALIDATION_TIME
     * (see params.h).
     *
     * Note that this error is not transient, target endpoint / XID
     * is definitely not known to peer.
     */
    AOS_ERROR_TARGET_UNKNOWN = -20,
```

```
/* The peer AppMW has (gracefully) closed a (virtual) connection */
AOS_ERROR_PEER_CLOSED_CONNECTION = -21,

/* The peer AppMW (role) has been deleted. Can be returned for any
 * call (e.g., AC shipment, communication) relating to a remote
 * AppMW. Note that this only applies when the target role is
 * explicitly deleted. If, for example, a target AppMW and/or AOS
 * crashes, an AOS_ERROR_IO is returned instead. The latter is no
 * transient error either, connections are really closed and need
 * resetting up when AOS crashes. However, in this case one would
 * generally expect (depending on AppMW implementation) the AppMW / AOS
 * to recover and publish its (new) communication endpoints in the
 * AppMW-level location or name service or whatever, which is not
 * likely when the AppMW ('s role) is really deleted. For this reason,
 * this error is important to inform AOS / AppMW that a role has been
 * deleted.
 *
 * This error only occurs _during_ a transaction or for already set up
 * communication channels. New connections to endpoints / XIDs of
 * this role will simply get an AOS_ERROR_TARGET_UNKNOWN (indicating
 * an unknown endpoint / XID).
 *
 * Note, PEER_ROLE_DELETED is returned only once per connection /
 * ACTP; subsequent usage of a channel / XID will return
 * AOS_ERROR_UNKNOWN_CONNID / AOS_ERROR_TARGET_UNKNOWN.
 */
AOS_ERROR_PEER_ROLE_DELETED = -22,

/*
 * 4. Security.
 */

/* For any access denied error except when covered by
 * METHOD_NOT_ALLOWED.
 * Used when an attempt is made to access a resource of which the
 * invoking role is not the owner.
 * ACCESS_DENIED is also returned when  when attempting to modify
 * an AOS_SYS segment.
 */
AOS_ERROR_ACCESS_DENIED = -23,

/* When the AppMW is not allowed to invoke a kernel primitive: the
 * role bitmap specifies 0 for this API call. */
AOS_ERROR_METHOD_NOT_ALLOWED = -24,

/* When a remote kernel cannot be authenticated */
AOS_ERROR_AUTHENTICATION = -25,

/* When two kernels cannot agree upon a security suite to use */
AOS_ERROR_SECSUITE_AGREEMENT = -26,

/* When the AppMW would exceed its assigned quotas */
AOS_ERROR_LIMIT_EXCEEDED = -27,

/* Error during transaction ID or cookie generation */
AOS_ERROR_RANDOMSOURCE = -28,
```

```
/*
 * 5. ACTP specific errors. When disk or network I/O errors occur,
 *    they should be reported rather than any of these errors.
 *    XID invalidation is discussed after the ACTP errors.
 */


/* When trying to ship an agent container that is not yet finalized
 *
 * XID is not invalidated, shipment can be retried (after finalize,
 * preferrably).
 */
AOS_ERROR_AC_NOT_FINALIZED = -29,

/* When the AC is too large (exceeds peer's internal limit). Note, this
 * is not a transient error due to lack of diskspace (see ..PEER_INTERNAL
 * errors below), the max. AC size is a hard limit in AOS.
 *
 * XID is invalidated.
 */
AOS_ERROR_ACTP_AC_TOO_LARGE = -30,

/* When the received AC is not in the right format
 *
 * XID is invalidated.
 */
AOS_ERROR_ACTP_AC_FORMAT = -31,

/* When the received AC is not correctly signed by the peer AOS.
 *
 * XID is invalidated.
 */
AOS_ERROR_ACTP_AC_VERIFY = -32,

/* When the receipt received from the shipping AOS after the AC has been
 * transmitted successfully is not correctly signed.
 *
 * XID is no longer valid, but AC is NOT removed at target (see below).
 */
AOS_ERROR_ACTP_RECEIPT = -33,

/* This error is depricated as of AOS API v5 rc3.
 * It may be used some time if we decide (v6?) that we require the
 * undocumented AC_PENDING_TIME parameter in params.h.
 * Do not implement this error code, or set AC_PENDING_TIME to
 * PARM_INFINITE so that it never occurs.
 */
AOS_ERROR_ACTP_TIMEOUT = -34,

/* Some error other than the above occurred, like AOS ran out of
 * disk. This error code is for transient errors - for example, when
 * disk ran out there may be space again when some other AC(s)
 * are migrated from the target AOS to another AOS. This error may
 * also be returned if the role for which the AC is intended has exceeded
 * a limit (e.g., MAX_ACs). Note that shipment of the AC should not be
 * postponed too long, as there may be a limit to the amount of time
```

```
     * within which an ACTP has to be initiated (see params.h).
     *
     * XID is not invalidated (see below), unless its invalidation
     * time expires (see XID_INVALIDATION_TIME in params.h).
     */
    AOS_ERROR_ACTP_PEER_TRANSIENT = -35,

    /* Some error other than the above occurred at the peer. This error
     * indicates a 'fatal' error, such as a disk I/O error (e.g., bad blocks)
     * at the peer.
     *
     * Note, this error and AOS_ERROR_ACTP_PEER_TRANSIENT are only returned
     * by ship_ac, wait_ac side may return e.g., AOS_ERROR_IO[_TRANSIENT],
     * instead.
     *
     * XID is invalidated.
     */
    AOS_ERROR_ACTP_PEER_INTERNAL = -36

    /* Notes on XID invalidation (tentative description):
     *
     * XIDs will be invalidated on both sides (on the receiving side,
     * any received parts of the AC will also be removed) when shipment
     * was succesful (i.e., a receipt has been returned), of when any of
     * the following errors occur:
     * AOS_ERROR_ACTP_AC_TOO_LARGE, AOS_ERROR_ACTP_AC_FORMAT,
     * AOS_ERROR_ACTP_AC_VERIFY, AOS_ERROR_ACTP_PEER_INTERNAL.
     *
     * Subsequent ship/wait_ac calls using a deallocated XID will yield
     * AOS_ERROR_UNKNOWN_XID on the waiting, and AOS_ERROR_TARGET_UNKNOWN on
     * the shipping side.
     *
     * The following errors are special, for the named reasons:
     *
     * AOS_ERROR_AC_NOT_FINALIZED: this is not an ACTP error really, it is
     * only a local error signalling an attempt to ship an unfinalized AC.
     * XID is not invalidated and the call may be retried.
     *
     * AOS_ERROR_ACTP_RECEIPT: receiving side sent a receipt, which means
     * it intends to keep and deliver the AC, even though the signature
     * was wrong from the sending AOS'es point of view.
     * Recovery has to take place at the AppMW level.
     *
     * AOS_ERROR_ACTP_PEER_TRANSIENT:
     * --> this error does NOT invalidate XID!
     *
     * AOS_ERROR_ACTP_PEER_TRANSIENT implies a transient error: the
     * ACTP request (using ship_ac or internal to AOS) may be retried
     * (using the same XID).
     *
     * Note that there is a timeout issue at the ship_ac side:
     * the XID may be invalidated if the first ACTP message
     * has not arrived within some AOS-defined timeframe (see params.h).
     * AC shipment attempts with a PEER_TRANSIENT error do not mark the XID/
     * ACTP as being in use, so the XID may be removed after the
     * AOS_ACTP_XID_INVALIDATION_TIME (see params.h) has expired/
```

```
     * This may occur even if multiple attempts to ship the AC were made
     * which returned a PEER_TRANSIENT error.
     */

};

/* Note on security:
 * The AOS_ERROR_UNKNOWN_COOKIE can be used by a local attacker to guess
 * valid cookies. The same holds for remote attackers and
 * ACTP_ERROR_XID_UNKNOWN in the AOS-level ACTP protocol.
 * We assume the kernel will be able to detect such
 * attacks and delete the attacker’s role or refuse communication with them.
 * Worse that could happen with guessing a XID is that a wrong AC is shipped
 * to, and actually received by, an AppMW. We assume that AppMW is able to
 * detect this, either by itself checking the sender’s endpoint as returned
 * by wait_ac, or by faulting on the content of the AC, such as AppMW-level
 * signatures of the TOC, or invalid AppMW-level invormation in an APP_SYS
 * segment.
 *
 * A similar problem existed with the local error AOS_ERROR_UNKNOWN_XID,
 * but we solved that by checking XIDs only relative to the role they were
 * assigned to (the coupling between role and XID is also necessary to allow
 * specification of XID=0 with wait_ac to wait for any XID that this role
 * created using prepare_wait_ac).
 *
 * In addition, we now introduced a maximum window of time within which the
 * ACTP has to be initiated (after that it stays bound to the initiating
 * AOS through the set-up base channel), to protect misuse of an (old) XID
 * by external attackers.
 */
```

CONTENTS

LIST OF FIGURES