Author Signature:	Date:
Approved by:	Signature:
Institute:	Date:

Keywords: ALMA, ACS, MACI, software|Released by: Mark Plesko, Gianluca Chiozzi|Signature:|

Institute: IJS, ESO	Date:

Management and Access Control Interface Specification

Software Specification

KGB Team, Jozef Stefan Institute

REVISION	DATE	AUTHOR	SECTIONS /PAGES AFFECTED
	REMARKS		
0.0	2001-03-16	Matej Sekoranja	All
	Applying ALMA template, changes in MACI		
0.1	2001-03-20	Klemen Žagar	All
	Added references. Changed CURL syntax. Introduced Startup and Immortal COBs. Classified three types of databases (CDB, LDB, PDB). Defined Configurable and Constructable interfaces. Extended Administrator interface (get_COB and release_COB notification). Extended Manager interface (get_DB_record). MACI-to-Naming Service mapping.		
0.2	2001-03-06	Klemen Žagar	3, 4, 4.1, 4.6, 5.5, 6
	Mark Plesko's comments taken into account. Chapters 3 and 4 renamed. Immortal COBs explained. Additional requirements for the Administrator interface added. Code-base manipulation methods of the Activator explained. Naming Service mapping rewritten.		
1.0	2001-04-01	Klemen Žagar	5, 4.2, 4.6, 4.7, 6
	Database removed from core MACI specification. Interface definitions moved to UML models. Inter-domain communication. Dependancy resolution on VxWorks. Naming Service mapping. Support for retrieving non-COB objects through the Manager.		
1.1	2003-11-11	Matej Šekoranja	All.
	Ported to new ACS 3.0 MACI IDL.		
1.2	2004-10-08	Klemen Žagar	5.1
	Manager does not have the "O" kind in the Naming Service.		
1.3	2004-11-15	Matej Šekoranja	4.6, 5
	Inter-domain Communication update.		
1.4	2005-09-08	Gianluca Chiozzi	5
	Added note about passing Component references directly between components.		

Table of Contents

This is the Management and Access Control Interface (MACI) Specification document. MACI is a part of Advanced Common Software (ACS) software.

Scope

The scope of this document is restricted to the discussion of MACI interfaces. The requirements for component behavior are a separate topic, discussed in Basic Control Interface Specification (BACI). MACI interfaces are static (in contrast to BACI interfaces, where only the design patterns are known in advance).
This document focuses on the MACI concepts and is not concerned with details such as definitions of interfaces and method signatures. Those can be found in the UML model[6] and the on-line documentation[7].

Glossary

http://www.alma.nrao.edu/development/computing/docs/joint/draft/Glossary.htm

References

1. ALMA-SW-NNNN, Revision 0.0, Basic Control Interface Specification, Gašper Tkaik.
2. CORBAservices – Property Service (chapter 13), OMG. Available at ftp://ftp.omg.org/pub/docs/formal/97-12-20.pdf.
3. CORBAservices – Naming Service (chapter 3), OMG. Available at ftp://ftp.omg.org/pub/docs/formal/97-12-10.pdf.
4. CORBA 2.3.1 Architecture and Specification, OMG. Available at: ftp://ftp.omg.org/pub/docs/formal/99-10-07.pdf
5. Interoperable Naming Service Specification, OMG. Available at ftp://ftp.omg.org/pub/docs/formal/00-11-01.pdf
6. ACS UML model. Available at http://kgb.ijs.si/KGB/Alma/Specs/ACS.mdl
7. ACS on-line documentation. Available at http://kgb.ijs.si/KGB/Alma/Docs/ACS-docs.tar.gz
8. ACS Logging and Archiving Specification. Available at http://kgb.ijs.si/KGB/Alma/Specs/Logging_and_Archiving.doc

In abstract terms, a ***control system*** can be viewed as a collection of ***services*** that enable the interaction between the ***controlled entities*** and the ***clients***. Controlled entities are mostly physical devices and other instrumentation. Clients are usually human users that wish to manipulate the controlled entities. When the control system is created as a collection of software components, two layers of functionality must be kept separate:

- ***services*** in the aforementioned sense of the word, which allow the interaction between the users and the controlled entities
- ***meta-services*** that ***manage*** the services themselves

Figuratively speaking, services provide content of the control system, while meta-services provide its form. Such a separation, not only on the conceptual, but also on the programmatic level is necessary for the following reasons:

- **Users**: there is a large number of service users, such as other services, GUI panels, business logic scripts etc. The meta-service users are limited, either in number or in possible ways of interaction with these services. Meta-service users are mainly administrator tools and, in a very limited and well-defined way, ordinary client programs.
- **Stability, performance and consistency**: A common meta-service framework for service management provides a uniform environment in which the services function. It also manages their lifecycle taking into account the state of the whole control system, which allows for performance optimization. By separating the management part out of the functional part of the service the system robustness increases, since the management code resides in one location only, not in every service separately. Moreover, the separate management is the hand-of-God that can terminate a hung service.
- **Maintainability and ease-of-use**: The management code can be located in one place and managed by a skilful programmer removing the need for service programmers to deal with management aspects of their services. Generic management tools can be written that allow the users to fine-tune or inspect the functioning of the control system.

To reiterate, the realization of these goals depends on the successful separation of the service and meta-service layers, which is possible due to the following characteristics of the control systems:

- **Ultimately, every control system is request-driven**. Therefore, lifecycle management can be separate from the service, since in principle, the service must exist for as long as the request executes and can be removed as soon as it completes. While such behavior is certainly not optimal, it demonstrates that a third party, privy both to the client requests and service replies, can manage the service's life cycle.
- **Ultimately, the control system controls physical objects**. This, in a sense provides a kind of persistence, since a service is only an interface to the existing controlled entity. The current state of such an entity together with a certain amount of static data in principle suffices to reconstruct the service. If any non-static persistence is necessary it is usually minimal. Therefore the service acts as a gateway and not as the process that progresses towards some final state and can thus be interrupted, restarted, removed etc. causing only the temporary unavailability of the service but not its subsequent failure when it resumes its function.
- **The clients are concerned only with the service functionality**. Meta-services are therefore free to provide the access to the functionality as they see fit. In other words, service locations are transparent and hidden to the ordinary (non-administrator) clients. It is assumed, moreover, that administrator clients that invoke meta-service functionality frequently issue sensible requests. In other words, the administrator clients must have higher security permissions and human operators synchronize their activity, so that the environment they create for the services is consistent, in contrast to services, which have their environment automatically managed by MACI.
- **The number of basic management operations is small and known in advance**. Contrast this with the service functionality that can be very varied and can be easily extended by simply adding new services. Services are also mostly independent of each other, but management operations are not.

Generally the services will be distributed – they will execute on a number of separate networked computers. Note that in the following discussion a service can also be a client to some other service. The distributed nature of the control system introduces these requirements into its design:

1. A client can gain access to a desired service, provided that its access rights are adequate.
2. A service can be relocated from one computer to another to improve performance of the control system or to recover from a failure, provided that the new host computer fulfils all service's requirements (e.g., controlled hardware device is physically connected to the computer).
3. If a service is not used for a longer period of time, it can be destroyed.
4. If a client requests a service that does not currently exist in the system, the required service is created on a computer where that specific service can exist.
5. If a newer version of a service is available, or if service configuration data changes, it must be possible to upgrade or reconfigure the service without interfering with its clients.
6. Users must have an overview and control over the current state of the system, including existing services and clients and the possibility to manually manage services.
7. Services distributed over disconnected networks must seamlessly federate when the networks are joined, i.e. the meta-services must be able to form federations.

Meta-services implementing the enumerated functionality are known collectively as ***Management and Access Control Interface*** (MACI) services. A service that is controlled by MACI is a ***component***.

Management and Access Control Interface is centered around CORBA Objects (components), but places no programmatic restrictions on them. Nevertheless the component can implement some of the interfaces declared by MACI in which case MACI use this additional functionality to improve the interaction between the component and MACI infrastructure. To proceed with discussion, let us first define and clarify the following concepts related to components.

Component

A component is a basic service provider in the control system. To the client a component is solely a certain CORBA interface that performs the declared functionality. To the management system, the component is an entity with a lifecycle that has to be managed. This document does not describe what mechanisms are used internally by the MACI implementation to control the component.
Regular components are instantiated on demand when they are first required, and released when they are no longer needed. There are tree other kinds of components:

- The ***startup components***, which are instantiated when the system (the Manager) is brought on-line. These components cannot be shut down, unless the entire system is shut down.

- The ***dynamic components***, which are components without any entries in the Configuration Database (CDB). They are instantiated manually by providing full component specification (*name, type, code,* and *container*).
- The ***immortal components***, which are instantiated when they are needed for the first time, but cannot be shut down afterwards. Consequentially, immortal components are not subject to MACI's life-cycle management, and references to them can be safely passed around (e.g., in Naming Service) without worrying that the components will be eventually shut down by MACI. As opposed to startup components, immortal components are activated only when needed, thus shortening the time required to start the control system.

CURLs and Domains

Since a component is essentially a service, it must be made available to the clients. In a distributed environment this means that every component must have a unique designation, which clients use to identify that component. As a meta-service, MACI then provides name resolution that from a well-formed component designation generates the component reference that can be returned to the client. Well-formed component designations are component URLs or ***CURL***s. A CURL is a hierarchical name implemented as a string of arbitrary length that consists of static prefix curl:, domain identifier and the component name. An example of a CURL might be curl://alma.nrao/antenna1/mount, representing the mount component for ALMA antenna number 1. Uniqueness is guaranteed by CURL as a whole and not by any of its components.
Domains allow hierarchical organization of names. It should be emphasized, however, that CURLs and URLs are not similar in the respect that URL name denotes the actual machine where a resource is to be found. CURL does not directly name any machine, port or other network location. CURL serves as a key that allows MACI to look up all the relevant network data from its data structures. A component can only be in one domain, which means that the domains partition the component space into disjoint sets. Although domains are logical partitions of the name space, they are restricted by the physical layout of the network: a single domain may not span a disconnected network (for instance two LANs, connected only by a dial-up line).
When a request is issued from domain A to domain B, we speak of inter-domain communication; if the communication is restricted to a single domain, we speak of intra-domain communication.
Both the domain and name components of the CURL are hierarchical with special characters introducing the hierarchy (slashes in the name component, and dots in the domain component).
In the root of every domain, the following components may be found:

- Manager – The Manager of the domain.
- CDB – Database Access Layer (DAL) component as an Configuration Database (CDB) for the domain. CDB contains configuration information for individual components, as well as MACI meta-services. CDB is to be defined in a separate document.
- PDB – Persistence Database component for the domain. PDB exposes the same interface as CDB, but it used for persisting internal state of objects. To be defined along with CDB (not implemented).

- Log - Centralized Logger [8] of the domain. Implements Telecom Logging Service's Log interface.

- NameService – The Naming Context of the domain. See page .

These names are reserved and no other component may be bound to them.
The set of characters used for CURLs is the same as that of URLs. It is recommended, however, that dots be used sparingly, as they have to be escaped in the Naming Service mapping.

Component Implementation

The knowledge that a component is a CORBA Object of a specific type suffices for the client. On the other hand MACI, which manages the components lifecycle, must also know where the actual component implementation resides. Upon request from the client, the piece of code that contains the component implementation is located, loaded and started. Information that MACI knows about the actual piece of code includes its name, the CORBA interface that is implemented and location from which it was loaded. This information is stored in the Configuration Database (CDB).
Such low-level management is highly centralized but is necessary for the purposes of reversible software upgrade and remote maintenance. A special object called Container, which serves as a component factory, manages the component implementation life cycle.
Aspects central to the component concept have been covered so far. MACI components that are responsible for realizing the described functionality are described below.

Client

Every client of a component service that is not itself a component may implement an interface called Client. The interface allows the client to act as a secure party in the communication with the components, to receive general-purpose string messages from the MACI components and to be notified when any change happens to the components that the client utilizes. Each client logs in to the MACI system before any other requests are made, and in turn it obtains a security token, which it must use in every subsequent request to the MACI. The log in and other requests are issued to the Manager, which serves as a portal to other services.

Administrator

Administrator is a special-purpose client that can monitor the functioning of the domain that it administers. Monitoring includes obtaining the status of the MACI components as well as notification about the availability of the component components. The administrator client is granted special access permissions. There is another condition tied to the Administrator interface: it must provide enough notifications about the state of the administrative domain so that the administrator client is always up-to-date with internal state of MACI components. In particular, notification of Manager's get_component and release_component requests must be issued, and login/logout notifications must supply all the necessary information that is required for authentication. This condition allows two uses:

- The administrator client can maintain a backup of MACI components' state, taking over MACI functions in case of MACI component's malfunction, thus avoiding single-point-of-failure.
- Discovery of dependency relationships between components in run-time. These relationships could then be displayed in a graphical user interface to the human user to aid in better understanding of the control system.

Manager

Manager is the central point of interaction between the components and the clients requesting their services. A Manager can manage more than one domain. Note that it is not necessary that the domain also corresponds to a single LAN. There may be a number of domains on a single LAN, but there cannot be a single domain spanning multiple LANs that are not directly connected (e.g., if they are only connected through dial-up links). Manager has the following functionality:

- It is the communication entry point. A client requesting a component service can do so by querying the manager. Security is tied to this functionality by requiring every client to pass the authorization protocol and subsequently tag every request to the manager with a security token generated by the manager upon successful authentication. Administrator clients may receive the details about the functioning of the manager, such as which clients are logged to the manager, which components are active etc.

Manager can serve as a broker for objects that were activated outside of MACI (non-components). It provides a mechanism for binding and resolving references to such objects.

- It performs as a name service, resolving CURLs into object references. Multiple CURLs can be resolved in one network call, either by enumerating them or using regular expressions. If a CURL is passed that is outside the current Manager's domain, the Manager forwards the request to the Manager closest to the destination domain to resolve it (see the chapter Inter-domain Communication).
- It delegates the component life cycle management to the Container object and therefore creates no components directly. However, it does maintain a list of all available Containers.
- The Manager uses the configuration database to retrieve relevant configuration for individual components in the domain, as well as locations of other Managers. Manager persists its internal state (such as references to activated components) in its portion of the persistent database.

- In environments with low security requirements maintains a hierarchy of CORBA Naming Service's Naming Contexts (see [3]). The Naming Contexts are arranged in such a way that they mirror the hierarchy of components in the domain, as well as relationships between domains (see chapter on Naming Service starting on page<ac:structured-macro ac:name="anchor" ac:schema-version="1" ac:macro-id="810d3ab2-e371-4c0e-b2e6-9997edb3a078"><ac:parameter ac:name="">_Hlt510238890</ac:parameter></ac:structured-macro><ac:structured-macro ac:name="anchor" ac:schema-version="1" ac:macro-id="d30bac01-15fc-44f5-be86-bc12dde9744c"><ac:parameter ac:name="">_Hlt510238891</ac:parameter></ac:structured-macro>).

Note: Besides losing security, component lifecycle management and on-demand activation are also not available when relying on the Naming Service alone to provide references to components.
Clients locate Managers by obtaining references to them from well-known locations, such as the web, configuration files and similar.
A Manager is named by a special syntax CURL and can return a reference to itself and other Managers.
Manager is the only interaction that clients have with MACI. Thus, neither component implementers nor GUI client developers need concern themselves with aspects of MACI other than the Manager.

Container

An Container serves as an agent of MACI that is installed on every computer in the control system. Every Container runs in its own process. Manager sends the Container the request to construct a specific component by passing it the name, type and the path of executable code of the component. The Container loads the executable code and begins executing it. If the dependant executables are not loaded automatically by the operating system (as is the case on the VxWorks platform), Container loads them prior to executing any code.
The Container also deactivates components, when so instructed by the Manager and is able to shutdown by disabling all components. Container maintains a list of all components it has activated and is able to return data about individual component's implementation.
Since the Container is a separate CORBA object, it can exist independently of the Manager, which provides another level of fault tolerance to the system. There must be at least one Container on every computer that is able to construct and export components. The means of bootstrapping Containers are implementation dependent and outside the scope of this document.
The Container is a self-contained object and must not hold any references to other objects except the components that it instantiated and the Manager of the Container's domain.
Container provides the components it hosts with an interface through which components can perform their MACI-related tasks, such as issuing requests to the Manager and activating other CORBA objects. This approach has two advantages:

- Reduced number of network calls because some requests can be resolved locally.
- Handling complex issues such as security and object activation.

When a Manager receives a request for a component that is not in the Manager's domain, the Manager forwards that request to the appropriate Manager, whose reference is obtained via central Naming Service. Central Naming Service holds references of all active domains where Naming Contexts hold references to the local domain Naming Service.

CURL-to-Naming Service Mapping

Naming Service (NS) is a collection of Naming Contexts (NC). Each NC holds named references to other objects (*bindings*), which may also be NCs. This allows one to build a hierarchy of object.
A name in the NC is composed of two strings: the id and the kind. The CURL-to-NS mapping uses three different values for kind:

- **D**: Containers for domain's root objects and sub-domains.
- **F**: Folders of objects.
- **O**: References to objects.
- Other: References to objects registered in a way other than through the Manager.

The mapping is as follows:

1. For every domain in the CURL hierarchy there exists a Naming Context (the *domain NC*) in the NS, which contains the following bindings:
    - NCs of sub-domains (kind "D").
    - For non-root domains, the NC of the parent domain. Such bindings' ID is "Parent" and kind is "D".

- Bindings to domain's root components, including the Manager. The kind of these bindings is "O".
- Manager's name is "Manager". Manager does not have a kind!
- Other objects. Kind must not be "D", "F" or "O".
- For each root object that has sub-objects, the domain's NC also contains a binding to a NC that contains the sub-objects. The kind of this binding is "F", and ID is the same as the CURL name of the object.

2. For every node in the component hierarchy there exists a NC (the *folder NC*). The NC contains:
   - A NC whose ID is "Domain" and whose kind is "D". This binding links to the domain NC of the domain to which the folder belongs.
   - A link to the parent's NC. The ID is "Parent", and the kind is "F".
   - NCs of subordinate hierarchical levels. The kind of these NCs is "F".
   - Other objects. Kind must not be "D", "F" or "O".
   - Components in the node's hierarchical level. The kind of these NCs is "O".

Bindings in a hierarchy of NCs are stringified according to the rules given in the Interoperable Naming Service specification [5], section 2.4.
Examples:

1. curl://sub2.sub.root/obj/subobj maps to sub.D/sub2.D/obj.F/subobj.O in the NS, resolved at the root domain's NC (notice the reversal of domain name components and the usage of dots and slashes).
2. Manager of domain sub.root (curl://sub.root/) is accessible as sub.D/Manager resolved at the root NC.

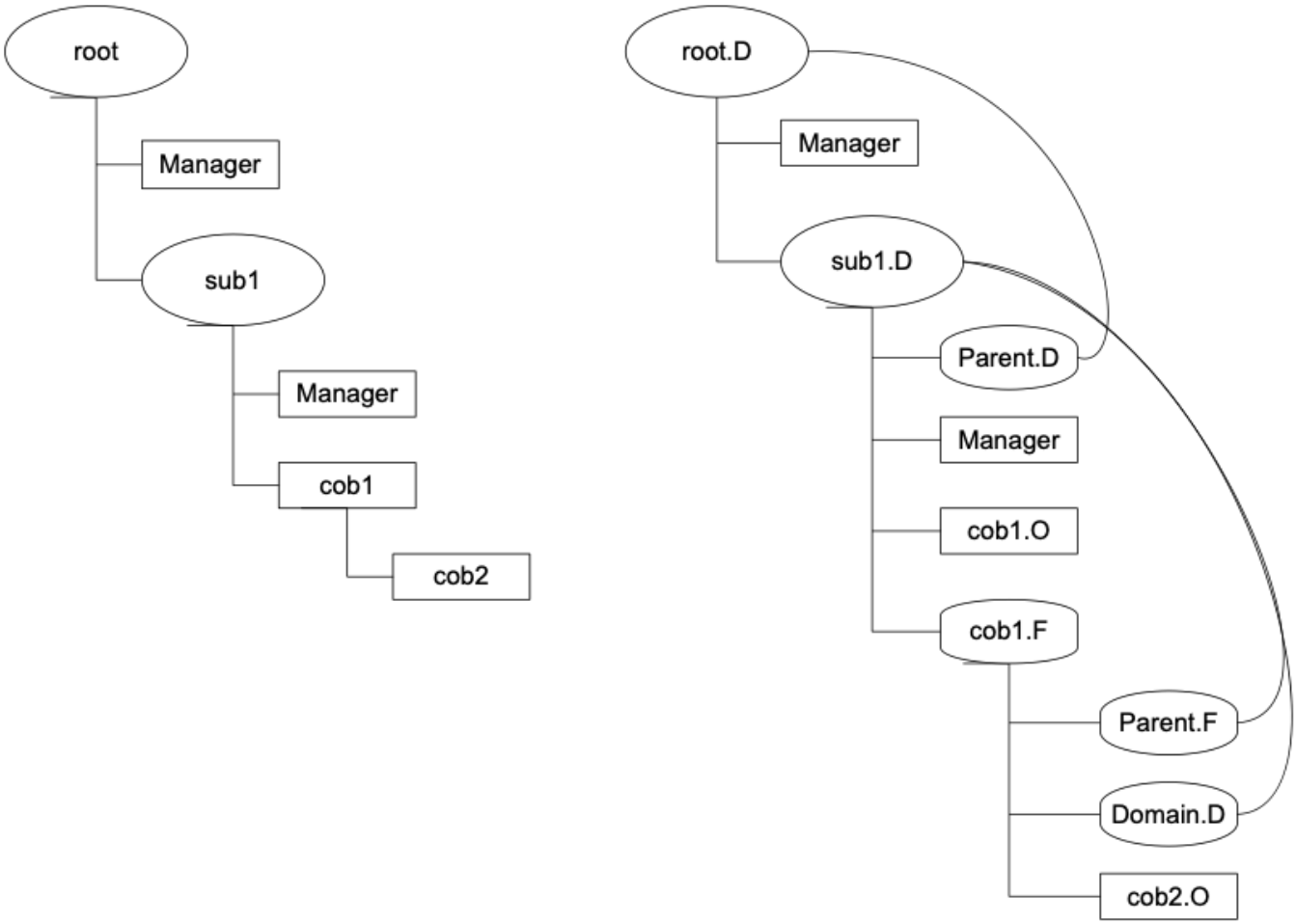

*Figure 1:**Left:** Hierarchy of domains (ovals) and components (rectangles). **Right:** Equivalent hierarchy stored in the Naming Service. Naming Contexts are denoted as ovals, whereas squares represent other objects. Lines denote the bindings.*

Accessing the Naming Service

Manager must be configured in such a way that when it issues an resolve_initial_references("NameService") standard CORBA call, it obtains the reference to the NC of the domain administered by the Manager.
Clients have two alternative ways to access the Naming Service:

1. By using resolve_initial_references.
2. By asking the Manager to retrieve a reference to the Naming Service via get_service("NameService").

In both cases, the domain's NC is returned.

Caveats when Using the Naming Service

Naming Service support is provided for greater interoperability with existing tools that use Naming Service for locating objects. Naming Service is not a replacement for Manager's get_component method. The following aspects have to be taken into account:

- Only activated components can be found in the Naming Service hierarchy.
- Requesting a non-activated component from the Naming Service results in an exception, whereas doing the same from the Manager would activate the component automatically.
- Manager guarantees that references to components acquired through get_component are valid until release_component is called. No such guarantee is made for references obtained from the Naming Service. (For immortal and startup component, the references are guaranteed to be valid even though they were acquired from the Naming Service.) This restriction stems from the fact that Manager takes care of component's life-cycle.
- Using Naming Service bypasses MACI security, allowing unauthorized clients to access components.
- Using Naming Service in components to locate other components prevents MACI's automatic dependency resolution mechanism from activating components in correct sequence, thus making system startup difficult (impossible in case of unintentional cyclic references).
- MACI cannot determine dependencies between clients and components if references to components were acquired through the Naming Service.
- For the same reason, it is not allowed to pass around directly references to Components, but only the Component's CURL.