XtreemOS Grid Operating System Component Interfaces Release V2.1

XtreemOS Consortium

March 2010

Contents

1 Introduction

This document describes the interfaces provided by the various XtreemOS components in XtreemOS 2.1 releases. After a description of the user interfaces in Section 2, we focus in the remainder of the document on the description of the internal interfaces, ie interfaces provided by a given component to the other XtreemOS components.

2 User Interfaces

XtreemOS provides two high-level user interfaces. The Xterior program offers a graphical user interface for people, while XOSAGA offers an high-level programming interface for applications.

2.1 The Xterior GUI

Xterior is a GUI application for easily managing files and jobs in a virtual organization. Xterior is implemented in Java, and uses the Java SAGA engine underneath to access XtreemOS functionality. Currently, Xterior only supports file management.

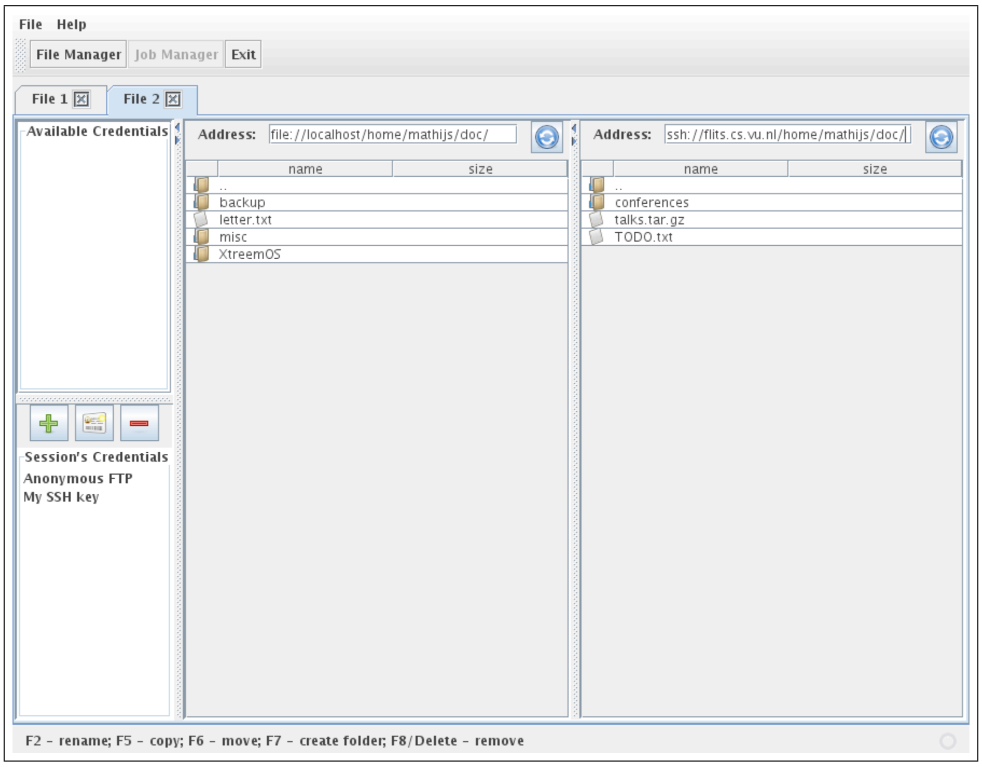

Figure 1: The Xterior file management interface

Figure 1 shows the file management interface of Xterior. Users can create multiple tabs, each showing the contents of two directories. Each tab uses its own SAGA session. The side pane on the left shows which credentials are used in each session. New credentials can be added to access different file systems. Files can be moved and copied between the two directories shown, either via keyboard shortcuts or by drag and drop. Users can also create, rename and delete directory entries. Double-clicking on a directory changes the current working directory, and double-clicking on a file opens it in the locally associated application.

2.2 The XOSAGA programming interfaces

XOSAGA is the Application Programming Interface (API) of XtreemOS. It is an XtreemOS-specific extension of the Simple API for Grid Applications (SAGA). The language-independent specification of SAGA [4] describes a high-level, object-oriented interface to the functionality of grid systems, like job submission, file management, security contexts etc. The second XtreemOS release contains XOSAGA implementations for three different programming languages: C++, Java, and Python.

2.2.1 C++ XOSAGA

The C++ implementation of XOSAGA extends the C++ reference implementation of SAGA with XtreemOS-specific *adaptors* that enable the use of XtreemOS functionality via the SAGA API. A detailed description of the C++ SAGA API can be found at `http://saga.cct.lsu.edu/cpp/apidoc/`.

The C++ XOSAGA implementation currently provides an XtreemOS-specific *file adaptor* to access XtreemFS volumes, and an XtreemOS-specific *context* for accessing user certificates. The following objects in the SAGA API therefore have an XtreemOS-specific implementation: `saga::ns_entry`, `saga::ns_directory`, `saga::file`, `saga::directory` and `saga::context`.

2.2.2 Java XOSAGA

The Java implementation of XOSAGA extends the Java SAGA implementation developed at the VU University in Amsterdam with XtreemOS-specific adaptors and APIs. Extensive documentation of the SAGA Java language binding is available online at `http://saga.cct.lsu.edu/java/apidoc/`.

Currently, Java XOSAGA offers a *file adaptor* to access XtreemFS volumes, an XtreemOS-specific *context adaptor* to access user certificates, and an *job adaptor* for job submission using the Application Execution Management (AEM) framework. In addition, Java XOSAGA provides an XtreemOS-specific API for resource reservation, which is implemented on top of AEM.

2.2.3 Python XOSAGA

The Python implementation of XOSAGA implements a preliminary version of the Python SAGA language binding. This language binding is under development, and will be candidate for future standardization. The current version can be found in the XtreemOS Subversion repository:

```
svn+ssh://scm.gforge.inria.fr/svn/xtreemos/grid/xosaga/python/trunk/doc/
```

Python XOSAGA is implemented as a wrapper around Java XOSAGA. It relies on Jython[1], a Python interpreter written in Java. Currently, Python XOSAGA offers the full SAGA API. XtreemOS-specific functionality is available via adaptors in Java XOSAGA, which offer access to XtreemFS, XtreemOS user certificates and job submission via AEM.

3 VO and Security Services

3.1 CDA Service

The Credential Distribution Authority service runs on a single core node. It uses the X-VOMS service.

The CDA generates XOS-Certificates, containing a user's public key and VO Attributes.

The core functionality of the CDA service is in two main classes. The Engine class is used to set up the constant parameters for creating certificates (such as the root certificate+private key, the CDA's own certificate+private key, certificate validity etc), and the XOSCertGenerator class is used to create an XOS-Certificate for a particular user request.

Other classes are used to implement the *communication* aspects of the CDA service, such as creating a service listener, parsing commands from the CDA client, etc. These classes are not described here.

The Engine class is essentially a high-level interface to the certificate generation code. The XOSCertGenerator and VOService classes are lower-level interfaces to this functionality, and will not be called directly by applications if the Engine class is used.

The constructor for the Engine class takes the credentials needed to sign a user's XOS-Certificate with the CDA's private key.

```
public Engine(PrivateKey cdaKey, X509Certificate cdaCert,
X509Certificate rootCACert, String signatureAlgorithm,
VOService voService,
int days, int hours, int minutes)
```

[1]http://www.jython.org

The method `generateXOSCert` takes the parameters username, voName and the Certificate Signing Request.

```
public X509Certificate generateXOSCert(String username, String voName,
PKCS10CertificationRequest userRequest)
```

The above two methods are the only ones that most applications will need to create XOS-Certificates.

Low-level classes used by the methods in the Engine class:

The VOService class is a no-constructor helper class containing two methods.

The method `authenticate` is used to authenticate a user in the X-VOMS database:

```
public boolean authenticate(String username, char[] password)
```

The method `getVOUser` retrieves the user's VO attributes from the X-VOMS database in a format that can be used by the XOSCertGenerator class:

```
public static VOUser getVOUser(String username, String voName)
```

The constructor for the XOSCertGenerator class takes the CDA's private key and public key certificate, and the name of the algorithm used for the signature applied to the certificate.

```
  public XosCertGenerator(PrivateKey cdaKey,
    X509Certificate cdaCert,
    String signatureAlgorithm)
```

The method `generateXOSCert` does the work of creating the XOS-Certificate for the user. The parameter `serial` is used for the serial number of the XOS-Certificate, hence it must be unique for this particular CDA. The duration of the certificate's validity can be expressed in the parameters `days`, `hours`, and `minutes` - one of these needs to be non-zero to specify a valid duration.

```
public X509Certificate generateXOSCert(VOUser user, long serial,
PublicKey userKey, int days, int hours, int minutes)
```

3.2 X-VOMS

In the first release, XVOMS provides two types of interfaces to other XtreemOS components: **UserUtil** - interfaces for managing users and **VOUtil** - interfaces for managing VOs. Currently, two XtreemOS components - CDA and VOLife use (part of) these interfaces to interact with XVOMS.

It should be pointed out that both CDA and VOLife have defined additional APIs to manage their interactions with XVOMS. For example, VOLife has defined additional non-native XVOMS APIs for managing VOs, subpassing those natively provided by XVOMS. When non-native XVOMS APIs are used, the interactions will not be subject to the native access control mechanisms and policies defined by XVOMS DB.

User and VO Management Interfaces in XVOMS UserUtil is a general interface for managing users in XVOMS. Users access all the VO management facilities via the XvomsSession interface:

- `XvomsSession()`: This empty constructor is for demonstration purpose.
- `XvomsSession(java.lang.String username, java.lang.String passMD5)`: Two parameters should be past to a XVOMS session.

XvomsSession is associated with a XvomsSessionContext, which are described as follows:

- `XvomsSessionContext(XvomsSession xsession)`: If the user in the given session is authenticated, the context (including user, voUtil, and userUtil objects) will be properly populated.

Within the context, each user operation is checked against the policies.

User Management APIs The followings are a list of APIs that can be used to manage users via XVOMS:

- `User addUser(java.lang.String realname, java.lang.String username, java.lang.String pass, java.lang.String des)`: creates a user entry in the users table of the XVOMS db.
- `User addUser(java.lang.String realname, java.lang.String username, java.lang.String pass, java.lang.String des, VORole vorole, Actor actor)`: creates user in the users table of the XVOMS db.
- `User getUser(java.lang.String username)`: gets a user object with a given username.

- `java.util.List getUsers(Actor actor)`: gets a list of users of a particular actor type.
- `java.util.List<User> getUsers(VO vo)`: returns a list of user objects of a given VO
- `java.util.List getUsers(VOGroup vogroup)`: returns a list of user objects of a given vogroup, more precisely, a given vogroup within a particular vo.
- `java.util.List getUsers(VORole vorole)`: returns a list of user objects of a given vorole, more precisely, a given vorole of a vogroup within a particular vo.
- `void removeUser(User user, VORole vorole)`: this API is *not being used and not being tested* in the first release.

VO Management APIs The followings are a list of APIs that can be used to manage VOs via XVOMS:

- `Action addAction(java.lang.String name)`: This method adds an action to the xvoms db.
- `Actor addActor(java.lang.String name, java.lang.String des)`: This method adds an actor into the system.
- `void addActor2User(User user, Actor actor)`: assigns an actor role to a user.
- `void addGroup2User(User user, VOGroup vogroup)`: gives a user a particular vogroup.
- `VOGroup addGroup2VO(VO vo, java.lang.String des)`: add a vogroup to a vo
- `RCA addRCA(java.lang.String name, java.lang.String desc, User owner)`: add a rca into the xvoms db.
- `Request addRequest(java.lang.String desc, java.lang.String type, VO targetedVO)`: adds a request to the user in the current context.
- `Resource addResource(VO vo, java.lang.String desc, RCA rca)`: adds a resource from a rca to an existing vo.
- `VORole addRole2Group(VOGroup group, java.lang.String des)`: add a vorole to a vogroup

- `void addRole2User(User user, VORole vorole)`: gives a user a particular vorole.
- `AccessControlRule addRule(java.lang.String desc, Actor actor, Action action, java.lang.String target)`: The targetObject means the table where the action is allowed to perform.
- `VO addVO(java.lang.String des, java.lang.String name, User owner)`: adds a vo to the X-VOMS db and set its owner to be the given owner.
- `void addVO2User(User user, VO vo)`: assigns the user to a vo, that is, a user joins the vo.
- `Actor getActor(java.lang.String name)`: gets an actor object given a name
- `java.lang.Object getObject(java.lang.String name, java.lang.Class aClass)`:
- `RCA getRCA(java.lang.String name)`: gets a RCA object given a name
- `java.util.List getRequests()`: gets a list of requests belonging to the current user in the session
- `java.util.List<Request> getRequests(VO vo)`: gets a list of requests belonging to a particular VO
- `VO getVO(java.lang.String gvid)`: gets a VO object given a gvid
- `java.util.List getVOAttributes(java.lang.String username)`: retrieve a list of VO attribute array (each array is an order set of VO attributes, such as [VOGroup, VORole]) of a user within a particular vo
- `java.util.List getVOAttributes(User user)`: returns a tuple in the form of [VO's GVID, VO description, vogroup, vorole, GGID] of a given user
- `java.util.List getVOAttributes(User user, VO vo)`: retrieves a list of VO attribute array (each array is an order set of VO attributes, such as [VOGroup, VORole]) of a user within a particular vo
- `java.util.List getVOGroups(VO vo)`: gets a list of vogroup of the given vo
- `java.util.List getVORoles(VOGroup vogroup)`: gets a list of vorole of the given vogroup

- `java.util.List<VO> getVOs()`: returns all VOs stored in the xvoms db
- `void prettyPrint(java.lang.String username)`: prints a user's VO attributes
- `void updateGroup(VOGroup group, java.lang.String des)`: updates an existing vogroup
- `void updateRCA(java.lang.String name, java.lang.String desc)`: update the description of the RCA.
- `void updateRole(VORole role, java.lang.String des)`: updates an existing vorole
- `void updateVO(java.lang.String gvid, java.lang.String des)`: updates an existing vo, given a gvid

Access Control Interfaces in XVOMS In XVOMS, all user management operations (i.e. add, remove, update, get) on user objects are under policy-based access control. Similarly, all VO management operations (i.e. add, remove, update, get) on objects (e.g. user, request, vo, voattributes) are also under policy-based access control. The access control policies are defined in the rules table (or the AccessControlRule objects) in the XVOMS db.

Roughly speaking, each policy contains three parts: an actor, an action, and a target. Finer-grained control can be further introduced: for example, by adding the validity of each policy and perhaps, or by introducing XACML like policies to take attributes of actor, action, target into account. Each policy is associated to an actor. Each user can be assigned to one (or more) actor. *In the current version, each user is only associated with one actor.*

All the access control enforcement are based on the two util interfaces: `VOUtil` and `UserUtil`, which only contain a subset of the functionalities offered by `VOUtilImpl` and `UserUtilImpll`. Via the XvomsSession, a user application, for example, the web frontend - VOLife, can only retrieve the interfaces `VOUtil` and `UserUtil`. The differences between `VOUtil` and `VOUtilImpl` (similarly between `UserUtil` and `UserUtilImpl`) allows XVOMS to control the level of functionalities that are available to a calling service.

However, if and only if the access goes via the util interfaces, it will be subject to the access control. Developers can get around the XVOMS access control by directly creating `VOUtilImpl` and `UserUtilImpll` as follows in their application:

```
new VOUtilImpl(null)
new UserUtilImpl(null)
```

By doing so, effectively, it means that the application itself has to incorporate authentication and access control into its application logic.

3.3 VOLifeCycle

VOLifeCycle is a web frontend for accessing VO management and security services including CDA, X-VOMS, VOPS, and RCA. In the second release, new features of CDA and X-VOMS are supported. The correct working of VOLifeCycle heavily relies on the stability of interfaces exposed by those security services.

Currently VOLifeCycle itself does not expose any development interface to other components. VOLifeCycle has a unified wrapper library around CDA, X-VOMS, VOPS and RCA client codes. As this is an interal library it may change without notice.

3.4 RCA Client

The service implementing the Resource Certification Authority client. This service runs on each node that is capable of providing services or resources to Virtual Organisations (VO). The service is the node's counterpart of the RCA server, providing a convenient way to store and access the local machine certificates, gather information on the resource (e.g. from the local ResourceMonitor service), and it also generates new public/private key pair, the former of which it then sends to the RCAServer for signing.

The class eu.xtreemos.xosd.security.rca.client.RCAClient implements the service's API. However, the methods should not be used directly by importing this class. Instead, the following interface should be used:

- `eu.xtreemos.xosd.services.SRCAClient` in other DIXI services, or
- `eu.xtreemos.xati.API.XRCAClient` in XATI client (the static methods of the class).

The interface exposes the following methods:

- `public String getMachineCertificateDetails()` opens the certificate currently stored locally and signed by RCA, and returns a string containing the details on the certificate.
- `public X509Certificate getMachineIdentityCertificate()` retrieves the node's machine identity certificate.
- `public RCASignedResponse getMachineAttributeCertificate(String vo)` retrieves the machine's attribute certificate.

- `public Boolean requestNewCertificate()` generates a new public and private key pair, then sends the public key for certification to the RCA server and obtains the signed attribute certificate.
- `public Boolean requestAttributeCertificate(String vo)` requests the resource's attribute certificate providing credentials for the given VO from the RCA server. It also installs the new certificate if the request succeeds.
- `public Boolean requestServiceCertificate(ArrayList<String> services)` requests the signing of the certificates to represent the given services to be run on the node.
- `public Integer applyForRegistration()` obtains or reuses own resource details, and send them to RCA derver for resource registration application.
- `public Integer pushVOAttributeCertificate(RCASignedResponse certResponse)` lets the RCA Server service push one or more machine's VO attribute certificates that can be installed and used on the local nod
- `public Integer removeVOAttributeCertificate(String vo)` lets the RCA Server remove an attribute VO certificate, notifying the client about removal from the VO.
- `public ResourceDescriptorRecord getResourceDescription()` retrieves the description currently set to the RCA client's instance.
- `public Boolean setResourceDescription(ResourceDescriptorRecord descriptor)` lets the resource descriptor used by the RCA client's instance to be assigned to a new value.

3.5 RCA Server

The service implementing the Resource Certification Authority server. The main purpose of the service is to sign the resource's identity certificate public key, and provide a signed attribute certificate to the resource.

The service implements the RCA server and the RCA database that keeps the collections of the registered resources.

A typical sequence of usage is as follows:

- The resource administrator runs an instance of the `RCAClient` service on the node that needs the registration and certification.
- The RCA Client by request or automatically calls `applyForRegistration` with the resource's details to add the resource data to the list of the resources pending for the registration.

- An authorised administrator within an organisation calls `confirmRegistration` using the ID of the resource to add to the list of the registered resources.
- The node can then use its instance of the `RCAClient` to create a private and public key pair and then call `requestCertificate` to obtain a signed identity certificate and a signed attribute certificate.
- The RCA can be added to (or removed from) a VO by calling `setVOMembership)`. This call influences all the registered resources that automatically become (or stop being) members as well.
- The clients can request the attribute certificate describing which VO they belong to using `requestVOCertificate(ResourceID, X509Certificate, String)`. Further, they can obtain a signed VO certificates for the VOs they are registered for using `requestVOCertificate`. They can also obtain service certificates using `requestServiceCertificate`.

The attribute certificate currently contains the following attributes:

- the number of CPUs in the node,
- the CPU speed
- the physical memory size, and
- a list of services running on the node.

The VO attribute certificate also contains the ID of the VO the resource is certified to belong to.

The configuration file **RCAServerConfig.conf** provides a way to configure the type of the attribute certificate (set attributeType to "V2" for attribute certificate, or to "V3" for attributes stored in extensions), the location of the trust store, the organisation details that form a part of the distinguished name (DM) of the issuer, etc.

The class eu.xtreemos.xosd.security.rca.server.RCAServer implements the service's API. However, the methods should not be used directly by importing this class. Instead, the following interface should be used:

- `eu.xtreemos.xosd.services.SRCAServer` in other DIXI services, or
- `eu.xtreemos.xati.API.XRCAServer` in XATI client (the static methods of the class).

The interface exposes the following methods:

- `public ArrayList<ResourceDescriptorRecord> getRegisteredResources()` returns a list of resource descriptions describing the resources listed in the RCA DB as registered.

- `public ArrayList<ResourceDescriptorRecord> getPendingResources()` returns a list of resource descriptions describing the resources listed in the RCA DB as pending for registration.
- `public Integer applyForRegistration(ResourceDescriptorRecord resource)` puts the resource on the list of resources that can be registered, but need to wait for an authorised administrator to confirm the registration using the confirmRegistration call before the resource can have its certificates signed by the RCA.
- `public Integer confirmRegistration(ResourceID id)` confirms the registration of a resource that has previously been applied for the registration using applyForRegistration.
- `public ResourceDescriptorRecord unregisterResource(ResourceID id)` removes the resource from the list of registered resources.
- `RCASignedResponse requestCertificate(ResourceID id, PKCS10CertificationRequest certRequest)` serves the client's request for signing the certificate.
- `requestVOCertificate(ResourceID id, X509Certificate certificate, String vo)` serves the client's request for signing a VO certificate.
- `RCASignedResponse requestServiceCertificate(ResourceID id, X509Certificate certificate, ArrayList<String> services)` serves the request for signing service certitificates. The new certificates will represent the services, one certificate per service in the input parameter collection. The passed certificate will be the base for the service certificates.
- `public Integer getResourceStatus(ResourceID id)` retrieves the current status of the resource according to the RCA DB.
- `public Integer setVOMembership(String vo, Boolean setMember)` is used to add the RCA into a VO or to remove it from a VO. All the registered resources are notified about any change.
- `public Integer notifyVOMembershipChange(ResourceID id, String vo, Boolean addition)` lets manipulate with the resource's membership of a VO.
- `public ArrayList<String> getVOList()` returns a list of the IDs of the VOs that the RCA is a member of.

3.6 VOPS

The Virtual Organisation Policy Service provides the means to store and manipulate the VO-level policies. It provides the policy decision point for the resource discovery used by the AEM's Resource Manager, and employs the RCA client for obtaining the resource candidate's public certificates.

The full documentation is available as javadoc in /Support/Documentation/ in the WP3.3 SVN.

The class eu.xtreemos.xosd.security.vops.VOPS exposes the following methods:

- `public Boolean registerVoAdmin(X509Certificate voAdminsCert)` adds certificate passed as an argument into a list of trusted certificates (VO admins list). This method is used as a base of access control.
- `public Boolean unregisterVoAdmin(Integer index)` removes certificate with specified index from a list of trusted certificates.
- `public String listVoAdmins()` lists registered VO administrators.
- `public String obtainFilterPolicyAEM(Object xosUserCert, String jsdlContent, String action)` obtains policy which will be used in resource discovery system as a filter (it will help to narrow down possible resource nodes).
- `public ResourceMatching verifyPolicyAEM(Object xos_cert, ResourceMatching resources, String action)` used by AEM framework to check if resources listed in comply with policies stored in VO policy storage PolicyFactory.listPolicies().
- `public ResourceMatching verifyPolicyCertRes(X509Certificate xos_cert, ResourceMatching resources)` is called by consequence of the verifyPolicyAEM(Object, ResourceMatching, String) method. Enforces policies - generates XACML request for each of the resource and checks it against policies residing in policy storage, see PolicyFactory.listPolicies().
- `public String createPolicyWithTarget(String policyID, String description, String target)` creates an empty policy containing target as provided.
- `public Boolean removePolicy(String policyId, X509Certificate userCtx)` removes the policy with policyId from policy storage.
- `public ArrayList<String> listPolicies(X509Certificate userCtx)` retrieves the list of currently available in the storage. Note that list of all policies can be very large. See also listFilteredPolicy(String) listPolicy(String).
- `public String listPolicy(String policyId, X509Certificate userCtx)` lists the specific policy with policyId. Returns XACML policy as String object.

- `public Boolean addPolicy(String xacmlPolicy, X509Certificate userCtx)` adds XACML policy into policy storage.
- `public String addRule(String ruleXML, String policyId)` adds a rule which is passed as XML string to the policy identified by policyId. Returns rule as String object. See also addXACMLRule(String, String), where rule is passed in a XACML format.
- `public String addXACMLRule(String ruleXACML, String policyId, X509Certificate userCtx)` adds rule which is passed as XML string to the policy identified by policyId. Returns rule as String object. See also addXACMLRule(String, String), where rule is passed in a XACML format.
- `public String listFilteredPolicy(String xacmlRequest)` returns a policy comprising rules which comply with the request passed as an argument.
- `public String listFilteredPolicyCert(X509Certificate targetSubjectCertificate)` lists policies which apply to certificate provided.
- `public String evaluateRequest(String xacmlRequest)` constructs XACML request from XML passed as request and applies xacml request against policies stored in policy storage.
- `public Boolean removeRuleFromPolicy(String ruleId, String policyId, X509Certificate userCtx)` removes rule from policy with specified policyId.
- `public Boolean writeBack(X509Certificate userCtx)` writes back policies from policy storage in PolicyFactory on to local disk.
- `public Boolean reloadVOPS(X509Certificate userCtx)` reloads all policies stored in policy storage.
- `public ArrayList<String> getSubjectAttributes()` retrieves the attributes that can be used in XACML policies, requests and rules.
- `public ArrayList<String> getResourceAttributes()` retrieves the attributes that can be used in XACML policies, requests and rules.
- `public ArrayList<String> getActionAttributes()` retrieves the attributes that can be used in XACML policies, requests and rules.

3.7 XOS-PAM

XOS-PAM is a Pluggable Authentication Module (PAM) to do mapping from VO users to local accounts. The module, named `pam_xos.so`, is developed as a shared module called by PAM-aware applications such as ExecManager in AEM.

To interact with `pam_xos.so`, PAM-aware applications can call the following standard PAM APIs, provided that `pam_xos.so` is configured to use in application-specific PAM configurations:

- `int pam_start(const char *service_name, const char *user, const struct pam_conv *pam_conversation, pam_handle_t **pamh)`: Initialization before invoking PAM module.
- `int pam_end(pam_handle_t *pamh, int pam_status)`: Termination of using PAM.
- `int pam_set_item(pam_handle_t *pamh, int item_type,const void *item)`: Setting parameters for given PAM module.
- `int pam_get_item(const pam_handle_t *pamh, int item_type, const void **item)`: Get a parameter from previous setting.
- `int pam_authenticate(pam_handle_t *pamh, int flags)`: Authentiate the user.
- `int pam_acct_mgmt(pam_handle_t *pamh, int flags)`: Account managment. In several cases, a virtual uid/gid for a VO user may be allocated in advance.
- `int pam_setcred(pam_handle_t *pamh, int flags)`: Tell the PAM module how to deal with the user's credential.
- `int pam_open_session(pam_handle_t *pamh, int flags)`:Open a PAM session.
- `int pam_close_session(pam_handle_t *pamh, int flags)`:close a PAM session.

3.8 XOS-SSH

XOS-SSH is a modied version of OpenSSH to support XtreemOS-specific certificate. For clients using XOS-SSH, no other special services need to be launched. The location of XOS-Certicates could be specied when starting the XOS-SSH client.

To coexist with standard SSH, XOS-SSH has attached suffix ”-xos” to all the SSH-related commands, such as ”scp-xos”, and the listen port is adjusted to 2222. However, all the usage of standard SSH is inherited by XOS-SSH. Therefore, users need not to pay additional study burden.

For resource nodes accepting connections from XOS-SSH clients, the dameon program (i.e. sshd) relies on XtreemOS PAM extension to dynamically allocate local user accounts for clients. The XOS-SSH extends OpenSSH by invoking XOS-PAM extension via standard PAM APIs.

3.9 AMS

The Account Mapping Service (AMS) plays the crucial roles of managing runtime user mappings and acting as local policy engine for VO access on the node.

To interact with AMS, applications need to link with AMS client libraries (`-lxos_ams`, `-lxos_common`, and `-lxos_db`) and include header files `include <xos_ams.h>` and `include <xos_protocol.h>`. The main interfaces to call AMS service are containing in following sections.

- **Mapping control:** The part of interfaces is included in `xos_ams.h`.
  - `int amsclient_usermapping(char *mapped_dn, char *mapped_vo,char *mapped_role, char *mapped_subgroup,AMS_GPASSWD * gpwd, AMS_GGROUPS * primarygrp)`:The function will pass the identity information to AMS server and told it to finish mapping.
  - `int amsclient_mappinginfo(char *mapped_dn, char *mapped_vo,char *mapped_subgroup, AMS_GPASSWD * gpwd,AMS_GGROUPS * primarygrp)`:Get mapping information according grid identity information.
  - `int amsclient_clearmapping(char *mapped_dn, char *mapped_vo,char *mapped_subgroup)`:Clear mapping information in local database according to grid identity.
  - `int amsclient_clearuidgid(int clear_uid, int clear_gid)`:Clear mapping information according to local uid / gid .
  - `int amsclient_invmappinginfo(char *username, int user_uid, char *groupname,int group_gid, AMS_GPASSWD * gpwd,AMS_GGROUPS * ggrp)`:The function will pass local user identity information and get back the grid identity information.
- **Policy control:** The part of interfaces is included in `xos_ams.h`.
  - `int amsclient_ruleput(struct rule_t *xrule, char **handler)`:put a rule into rule database.
  - `int amsclient_getrulebykey(unsigned int skey, struct rule_t **xrule,char **handler)`:get a rule by rule key.
  - `int amsclient_getrulebyhandler(char *skey, struct rule_t **xrule)`:get a rule by rule handler.

    - `int amsclient_rulesearch(struct rule_t *xrule, char **handler)`:search a given rule.

- **Container control:** The part of interfaces is included in `xos_cgrp.h`. These interfaces require support from kernel virtualization facilities (`cgroups` and `namespace`). To call these interfaces, additional shared library need to be linked is `-lxos_lxc`.

    - `int hd_cgroup_mountfs(char *subsys)`: setup a specific subsystem.
    - `int hd_cgroup_umountfs(char *subsys);`:uninstall given subsystem.
    - `int hd_cgroup_create(char *name, char *subsys);`:create a container in given subsystem.
    - `int hd_cgroup_destroy(char *name, char *subsys)`:destroy given container in given subsystem.

- **Protocol:** The part of interfaces is included in `xos_protocol.h`.

    - `int amsclient_connect_open()`:open the connection with AMS server.
    - `int amsclient_connect_close(int sockfd)`:close the connection with AMS server.
    - `int amsclient_connect_query(int sockfd, char *message, int *message_len)`:Send query message to AMS server.
    - `int amsclient_packetmsg(char *message, GPASSWD * gpwd, GGROUPS * ggrp,int ret)` and `amsclient_unpacketmsg()`:packaging function of mapping information.
    - `int amscient_unpacketmsg_rule(char *message, struct rule_t *xrule,char *handler, int ret)`: unpackaging function for policy rule.

3.10 Credstore

`libcredstore`: a credential storage abstraction.

The component 'credential storage library' (a.k.a `libcredstore`) has been developed initially for the mobile flavour of Xtreem-OS but it may also be used within other Xtreem-OS flavours or even in other projects as necessary.

3.10.1 Introduction

The main application of the `libcredstore` is to abstract the use of "Key Retention Service (KRS)" needed by the current implementation of XtreemOS

VO support, and allows for the use of other replacements without modifying or recompiling applications. This feature is very important in embedded devices, as recompiling and replacing the kernel may be a very complex or impossible task.

- For instance, in the Nokia devices based on the Maemo distribution (e.g. N800 and N810) users can easily install applications with a single click, but supporting KRS would require that the user re-flashes the mobile device to change the kernel, which should be avoided for security reasons.

libcredstore by default **uses the "Key Retention Service" if it's supported by the runtime kernel**, with a new feature: it **compresses and decompresses keys on the fly**. It does not require that libkeyutils package is installed, because it invokes syscalls directly. If the kernel does not support KRS, libcredstore detects that syscall faults and uses a replacement consisting in daemons executed in user space.

The usage of the API is very simple: a program just needs to include `credstore.h` in the code and invoke `get_creds_store_funcs()`. This function returns a `struct creds_store_funcs` with pointers to functions (e.g. `delete_cred` member is used to revoke a key, `store_cred` is used to save the key, `retrieve_cred` to get the key...). There also exists a `get_creds_store_funcs_by_impl`, in order to retrieve a specific implementation of the credstore. Current implementations include:

- krs: the usual Key Retention Service-based implementation.
- zkrs: is the same as krs, but in this case `store_cred` compresses the key and `retrieve_cred` uncompresses it. Compress ratio gain is variable, but as a reference, the size of `/etc/xos/creds/config2.pem` example credential is reduced to 69% of the original size.
- uskeystore: (user-space keytore)] this implementation is based on a daemon per user (or per session). It is designed to make it easier to modify the implementation to use a system daemon, instead of a daemon per user, or to adapt the module to use e.g. the GNOME keyring[2].

Current implementation of `get_creds_store_funcs` returns zkrs pointers if KRS is available, otherwise returns `uskeystore` function pointers.

Important: To ease debugging, the Makefile compiles `libcredstore` with the `USE_CREDSTORE_IMPL_ENV` variable defined. With this compilation option, `get_creds_store_funcs` first tests if environment variable

[2]The SVN version of gnome-keyring has exciting features like a PCKCS#11 module or integration with ssh-agent (see http://live.gnome.org/GnomeKeyring/Cryptoki)

`CREDSTORE_IMPL` is defined; in this case, it invokes `get_creds_store_funcs_impl` with the variable value. However, it is safer for a production-strength library to be compiled without this option (leaving the variable undefined).

3.10.2 struct_creds_store_funcs documentation

The goal of a credstore is to provide a credentials cache, to implement a Single Sign On mechanism, independent of the real storage system used (e.g. Kernel Key Retention Service).

A credstore requires the following **features**:

- Applications can **store a key in a credstore common to all applications of the same user**, even if they have been started using different login sessions.
- Applications can **store a key in a session credstore**, that is, only accessible to child processes of an initial process; the key is removed when this initial process ends. This feature is useful, for example, to start a shell and that key is only accessible to processes launched from this shell, during shell life. Unfortunately, security is only guaranteed with krs and zkrs implementations without additions, because any user application may spy environment and memory of any other application of the same user, reading /proc filesystem info, except if program permissions includes "execute" but not "read". Indeed, this security problem is present also with KRS, because after key is read, it is present in process memory, and any process can use ptrace to control and spy any other process of the same user if it has "read" permission. The real usefulness of this feature is to store a key only while the session is running and without affecting other sessions, that is, session key isolation is more like a "key namespace" than a full-fledged security feature.
- Applications can **remove a key** from credstore, or **set a timeout** after which the key in the cache is no longer valid.
- Only **one key by credstore (user or session) is supported**, but application can label the key with a configuration name, to known if the key currently stored in the cache is the key that user application want to use. If the application requests a key from other configuration than the one stored in cache, current key in cache is discarded and the new key will overwrite it. To limit users to store only a credential is less comfortable than to allow for multiple credentials simultaneously, but this policy is more secure because only one key is exposed at a time (user has the alternative of running different sessions each one with a single key). This is specially true with embedded devices, where few

(if any) unrelated processes run simultaneously. Other reason to allow only one credential is that KRS space is limited to only 10,000 bytes per user, but unfortunately if the user launches several simultaneous sessions this limit is also easily reached.

These are the function pointer members of `struct_creds_store_funcs`[3]:

`int store_cred(char *data_in_pem_format);`
This functions stores the credential that the user passes in PEM format (base64 encoded). If other credential is already stored, the new credential will overwrite it. Returns 0 on success, -1 on error.

`char* retrieve_cred (char *config_name);`
If `config_name` is NULL, this function returns the credential stored in the credstore (e.g. in Key Retention Service). If config_name is not NULL, first it checks if `config_name` is the current configuration name in the credstore; returns NULL if they are different.

`int check_cred_is_available (char *config_name);`
This function is similar to `retrieve_cred`, but returns 1 if the credential to this `config_name` is available, 0 otherwise (instead of the credential/NULL).

`int set_cred_timeout (int timeout);`
This function sets a timeout (in seconds) after which the credential is purged from the credstore. A zero value cancels the timeout. Returns 0 on success, -1 on error.

`int delete_cred ();`
Purges the credential from the credstore. Returns 0 on success, -1 on error.

`char * get_current_cred_configname ();`
This function returns the current configuration name of the key stored in the credstore. The initial value is “default”. Returns NULL on error.

`int set_current_cred_configname (char *config_name);`
This functions sets the current configuration name of the key stored in the credstore. Returns 0 on success, -1 on error.

`char * join_new_session ();`

[3]This API is not considered stable yet, and could be changed in the future if more functionality is needed from it.

Up to now, all functions operate over the current user credstore. This function creates a new credstore session and all successive new calls of the process and its children will use this new credstore session. It is important in this case to call `exit_session` at the end of the session (although currently `krs` and `uskeystore` do not need it because the session will be killed anyway when the program ends).

In the KRS implementation, this function invokes the syscall to start a new session. In the `uskeystore` implementation, this functions starts a new daemon using a fork and defines an environment variable with the address of the socket.

The function returns an environment value that the user must use with `putenv`, if launching other processes within the same session. The KRS implementation returns nothing, but `uskeystore` needs to modify the environment because new session processes use `XOS_CREDSTORE_SOCKET` to locate the socket of the session credstore daemon and to read the cookie and file handler required to authenticate.

`int exit_session ();`

If a user program calls `join_new_session`, it must call this function to end the session. In the KRS implementation this function does nothing (since the keystore session ends when the program that started the session ends). In the `uskeystore` implementation, this function kills the session credstore daemon, but this call is not needed in Linux because `join_new_session` calls `prctl` to establish that the credstore daemon receives a `SIGTERM` signal when the parent process dies. Returns 0 on success, -1 on error.

`int join_default_user_session ();`

If a program changes the UID, it must call this function to change to the new user credstore. In the KRS implementation, this function invokes the syscall to change the keystore. In the `uskeystore` implementation, it launches a new `xos_credstored` daemon. Returns 0 on success, -1 on error.

Important: If an application changes user, it must change the UID, not just the EUID, because in KRS the UID is used to select the user keyring[4]. With KRS, the FSUID (File system UID, which initially has the same value as the EUID) is used to assign the owner of a new created entry. If the program does not need to recover root privileges, the best solution is to change both the UID and the EUID. If the program needs to recover root privileges, the best option is continue with EUID=0, and change the UID with `setreuid`, and change the FSUID with `setfsuid`.

[4]With the `uskeystore` implementation, actually EUID is used to authenticate the socket instead of UID, but this is transparent to developers, because the module internally swaps UID and EUID before connecting with the daemon and restores them afterwards.

3.10.3 Utilities

`xos_dumpcred [<configname>]`

If no configname is specified, it returns the credential stored in credstore (if any). If a configname is specified, first it checks if configname is the current configuration name of the credstore; then returns nothing if they are different.

`xos_storecred [<filename>[<configname>]]`

If a filename is specified, this saves in credstore the contents of the file. If configname is specified, it also fills in the current configuration name with configname. If no filename is specified, it will read the content from standard input.

`xos_deletecred`

Delete current credential from credstore, if any.

`xos_settimeout <seconds>`

Set a timeout (in seconds) after which the credential is purged from credstore. Setting a new timeout always overwrites the previous timeout. Zero value cancels the timeout.

`xos_currentconfing`

Returns the configuration name assigned to credstore. The initial value is "default".

`xos_credstored`

This executable is the daemon that implements user-space credstore (uskeystore) implementation. The session credstore does not use this daemon: is implemented as a fork in the the process that invokes join_new_session().

The daemon is automatically launched when invoking join_default_user_session() or store_cred(), but can be started manually by the user also. It creates a Unix socket in the user's home directory, with name "socket_xoscredstore_<uid>". Only programs with the same uid as xos_credstored can use the socket.

Xos_credstored daemons run until machine is poweroff, because they are shared within different sessions of the same user and credentials survives the session that stores it. But daemon died if timeout end or delete_cred is invoked.

Xos_credstored tests that the daemon is not already running. The test consist in a special request to the daemon using the Unix socket.

3.10.4 Installation & Testing

Source package includes a INSTALL file with instructions.

Currently, libcredstore and associate utilities are included in xosmd-vosupport package, but can be compiled and installed independently of the other parts of the package. It is possible that in future releases they will be isolated in a independent package. The only dependency of xosmd-vosupport that is not required by libcredstore is libpam. The other requisite to compile the package is libz.

To install the utilities and library, just run "make install-credstore" (or run "make package-credstore.tgz to create a package with the binaries to install). Software is installed in /usr/local; it may be necessary to invoke ldconfig or to add /usr/local/lib to /etc/ld.so.conf.

Directory test/ includes two programs to test libcredstore: test_credstore and run_credstore_session. This test programs are also installed with "make install-credstore".

test_credstore first changes the configuration name to "configtest" and then retrieves the configuration name; it should show "configtest". Then it stores "key value" and invokes check_cred_is_available and retrieve_cred, with "configtest" as configuration name; it should show "true" and "key value". This test is repeated but using "fake" as the configuration name; it should show "false". And then it does the same without any configuration name: it should show "true" and "key value". Afterward, it deletes the credential and invokes check_cred_is_available and retrieve_cred; it should return "false". At last, it checks the timeout: first, it inserts a new value (remember that credential was deleted in last step) and invokes set_cred_timeout with a timeout of 4 seconds; then invokes check_cred_is_available and retrieve_cred, after sleeping 2 seconds and then after 2 additional seconds: it must show "true", "key value", "false".

run_credstore_session is a utility that launches a shell between a call to join_new_session and exit_session; that is, a new credstore is running until user runs "exit". With the uskeystore implementation, a "ps" will reveal two instances of run_credstore_session: the second one is the credstore daemon.

3.10.5 Appendix A: Adventages of KRS over User Space KeyStore

- uskeystore requires a daemon by user/session; KRS doesn't.
- KRS session keystore is hidden from other programs of the same user; with uskeystore this requires more changes, because /proc filesystem allows access to the environment of the rest of program by the same user and the cookie to access credstore is passed as an environment variable. Indeed, the /proc filesystem also allows to access the memory space of other programs and search for the secret key; this is a problem

also with KRS, but only while the program uses the credential, if the program fills the key with zeroes of random data after using it.

- uskeystore has not been tested against race conditions.
- uskeystore is sensible to DoS attacks (but only by the user that stores the cred). The uskeystore implementation only proccesses one request at a time; thus, a single hostile process may block the daemon. The implementation establishes a short timeout after a cancel operation. This model may change in a future.
- KRS stores keys in kernel memory, not accessible from user space programs and not swappable.

For all this reasons, it is recommended to use KRS if it's available and if the 10,000 byte quota per user is not a problem.

3.10.6 Appendix B: Caveats

- Security of any credential cache is weak against malicious applications from the same user (e.g. a ojan). Although a program with setuid/setgid or without read permission could stop ptrace, the problem is that applications that use the credentials cache need the same protection. Our implementation does "best effort" to secure session credstore: authentication between client and credstore checks that both run with the same UID, that client provides a secret cookie and a open file handler inhered from the application that started the session. The open file handler is created with socketpair in join_new_session function, and authentication is based in these three properties:
  - open handlers are heritable: they survive fork & execve.
  - open handlers can be transferred using a Unix socket.
  - with SO_PEERCRED getsockopt, session daemon can test that received handler is one socket created by its parent (retrieved credential includes PID of the socket creator); only descendants inherit this open socket. Of course this requires that file handler creator don't accept connections; otherwise any application may obtain a open socket that returns the same SO_PEERCRED credentials than file handler created with socketpair and inhered by all children.
- Operations storing credential and setting configuration name should be joined in one atomic operation to avoid race conditions.
- libcredstore features are limited to the ones that KRS offers. More advanced features, like allowing only specific registered applications to

read credstore, or like setting session credstores as read-only, are not possible with KRS, because applications could use the KRS API directly to circumvent libcredstore controls. Perhaps a future implementation of libcredstore would include functions or options not mandatory to implement, and get_creds_store_funcs function would have a new parameter to set the features that the returned implementation must provide.

- A good alternative to libcredstore and libxos_getcred could be a PKCS#11 module. This module may be implemented as a daemon that is the only client of a session credstore (a better option is that libcredstore also offers credstore private to process), and a library that accesses this daemon using a socket. Of course, it is simpler that the credstore implements directly the PCKS#11 module (as in the coming release of gnome-keyring) but with the KRS implementation this implies that applications that have access to credstore also have access to key directly.

More info about the specifications of this simple API may be found at D2.3.4, section 2.7.1 and in the URL: `http://xtreemos.wiki.irisa.fr/tiki-index.php?page=WP2.3+libcredstore&highlight=libcredstore`.

4 AEM

4.1 Job directory

Job Directory provides to the Job Manager the means to store and retrieve the information on the jobs, their owner and execution location. It uses ADS for the storage and querying.

The best way to see the expanded interfaces is inside /Support/Documentation/ in AEM SVN.

The class eu.xtreemos.xosd.jobDirectory.JobDirectory exposes the following methods:

- `public void addJobToJobDir(String jobId, CommunicationAddress addrJobMng)` adds the job to the jobDirectory

- `public void removeJob (String jobId, String userId)` Removes the job from the jobDirectory.

- `public CommunicationAddress getJobAddr(String jobId)` Gets the address of the job manager where the job is registered.

- `public ArrayList<CommunicationAddress> getJobAddrList(ArrayList<String> jobId)` Gets the list of address of the jobs managers where the jobs are registered.

- `public Hashtable<String, CommunicationAddress> getUserJobs(String userId)` Gets the list of jobs (and adresses) of the user.

- `public String getJobIDs()` Gets the list of JobIds of the Jobdirectory (Not supported with ADS)

4.2 Job Manager

The Job Manager includes all interfaces related to job management. The interfaces are automatically exported to XATI and C-XATI interfaces to be used by other components.

The best way to see the expanded interfaces is in /Support/Documentation/ inside AEM SVN.

- `public String createJob(String jsdlFile, Boolean startJob, String reservationID, X509Certificate userCtx)` (Synchronous) Creates a job in the AEM based on the JSDL description. The job can be automatically scheduled or just created, depending on the value of startJob. If a reservationId is provided, the job will be scheduled on that reservation. Otherwise, a negotiation/reservation process will be started when the job will be scheduled based on resource requirements and scheduling hints. If calls to VOPS are enabled, policy enforcement gets into the picture.

- `public void runJob(String jobId, String reservationID, X509Certificate userCtx)` (Synchronous) Makes call to resource manager to get all resources and continues in callback. If I am not an owner, I query the job directory for job address and make runJob call to that node.

- `public Integer runJobRes(String jobId, String reservationID, CommunicationAddress resourceID, X509Certificate userCtx)` (Synchronous) Starts the scheduling process of a previously created job. JobId must be a valid jobId in the system. Starts the scheduling process of a previously created job. JobId must be a valid jobId in the system. It will get a ResourceID, (that should be valid in the ReservationID provided), and run the job on that resource.

- `public void jobControl(String jobId, Integer ctrOp, X509Certificate userCtx)` Apply the operation Control to the specific jobId

- `public void exitJob(String jobId, Integer exitValue, X509Certificate userCtx)` The job identified by the jobId is finished immediatly (all the processes of the job) with the exit code provided.

- `public String getJobsInfo(ArrayList<String> jobIds, Integer flags, String infoLevel, ArrayList<String>)` Returns the monitoring information of the requested jobs. Any user of the VO can access monitoring data of the job, but if he is not the owner, access will be restricted to unbuffered system metrics. XML Output, please check Javadoc and XATI/XATICA samples to see how the user can process them.

- `public String getJobInfo(String jobId, Integer flags, String infoLevel, ArrayList<String> metrics, X509Certificate userCtx)` Returns the monitoring information of the given job (Deprecated, use getJobsInfo instead)

- `public ArrayList<String> getJobsUser(String userId, X509Certificate userCtx)` Returns the monitoring information of the given job

- `public void sendEvent(String jobId, Integer signal, Integer operation, ArrayList<String> list, X509Certificate userCtx)` Sends an event to a job

- `public Integer jobWait(String jobId, X509Certificate userCtx)` Blocks the calling process until the job indicated finishes.

- `public void createProcess(String jobId, String JSDL, String reservationId, CommunicationAddress resource, X509Certificate userCtx)` Creates a process and binds it to the specified jobId.

- `public void addDependenceUp(String jobId, String FromJobId, String TAG, X509Certificate userCtx)` Adds a tagged dependence between two jobs. Direction Up.

- `public void addDependenceDown(String jobId, String toJobId, String TAG, X509Certificate userCtx)` Adds a tagged dependence between two jobs. Direction Down.

- `public void deleteDependenceUp(String jobId, String FromJobId, String TAG, X509Certificate userCtx)` Removes a tagged dependence between two jobs. Direction Up.

- `public void deleteDependenceDown(String jobId, String toJobId, String TAG, X509Certificate userCtx)` Removes a tagged dependence between two jobs. Direction Down.

- `public void addDependence(String FromJobId, String toJobId, String TAG, X509Certificate userCtx)` This function adds a new job dependence. Dependence will be bidirectional: fromJobId — DOWN–⟩ toJobId and fromJobId ⟨– UP — toJobId. Take into account that inverting parameters makes a difference because of the different up/down lists. Unidirectional dependences might be added with other methods specified above. We support sets of dependencies identified by TAG, to be able to use them for different purposes. AEM won't interpret these TAGS, just group dependencies of jobs based on them. AEM won't check cycles in job dependencies. It is user/job responsibility.

- `public void deleteDependence(String FromJobId, String ToJobId, String TAG,X509Certificate userCtx)` Deletes an existing dependece between two specific jobs.

- `public ArrayList<String> getListOfDependencies(String jobID,String TAG, Integer levels, Integer directrion, X509Certificate userCtx)` Returns the list of jobs that have a dependence FROM jobID or TO jobID.

- `public ArrayList<MetricsDesc> getJobMetrics(String jobId, X509Certificate userCtx)` Returns the list of available metrics for a specific job, both system and user defined.

- `public Integer setMetricValue(String jobId, String metricName, CommunicationAddress resourceID, Integer pid, String value, X509Certificate userCtx)` Sets the value of a Metric. A metric will be an user-defined attribute of the job. Not all the attributes can be set, for instance the user time or the status are set by the system, not by the user.

- `public Integer setMonitorBuffering(String jobId, String metricName, CommunicationAddress resourceID, Integer pid, Integer flags, X509Certificate userCtx)` Switches on and off buffering for the specified metric. With buffering on, multiple values of a metric are returned, and its timestamps represent the time when the value was changed. Only metrics defined as "bufferable" on creation can be buffered.

- `public Integer addJobMetric(String jobId, MetricsDesc metric, X509Certificate userCtx)` Adds a new user defined metric to the job. Afterwards, user might give values to it through the setMetricValue interface and get them with getJobInfo. Metrics are checked for correctness before insertion. It might communicate with other services and XOSDs if needed.

- `public Integer removeJobMetric(String jobId, String metricName, X509Certificate userCtx)` Removes a user defined metric from the job. Data associated to the metric is also erased. It might communicate with other services and XOSDs if needed.

4.3 Execution Manager

The Exec Manager includes all interfaces related to job execution management. The interfaces are automatically exported to XATI and C-XATI interfaces to be used by other components.

The best way to see the expanded interfaces is inside /Support/Documentation/ in AEM SVN.

- `public ArrayList<Integer> getProcsJob(String jobId)` returns the list of pids of the specified job.
- `public String getProcsInfo(String jobId)` Returns text information about job's processes.
- `public void exitJob(String jobId, Integer exitValue)` send a Term (15) signal to all the processes of the job running in this node waits 5 seconds and sends kill (9) signal to the processes still alive
- `public String getJobSelf(Integer pid)` Return the JobId of the calling process (identified by its pid)
- `public String getJobsInfoByResource(CommunicationAddress resource, X509Certificate certificate)` Return the information of the jobs running in this resource
- `public String getJobsInfoResource(X509Certificate certificate)` Return the information of the jobs running in this resource. Certificate is checked by jobMng on a job basis.

4.4 Resource Manager

A class implementing the Resource Manager service. The service collects the available computation node and enables the node selection by resource queries expressed as a part of a job description. The job descriptions are formed as XMLs using the JSDL schema. Resource Manager queries nodes' local Resource Monitor service to obtain the resource descriptors formed as XMLs using the GLUE v.1.2 schema.

The class eu.xtreemos.xosd.resmng.ResMng exposes the following methods:

- `public ArrayList<CommunicationAddress> getResources(String query, X509Certificate userCtx, Integer howMany)` retrieve a collection of resources that match the job's resource demands.
- `public Hashtable<String, String> getResInfo(CommunicationAddress resource, X509Certificate userCtx)` returns the monitoring information associated with the resource.

- `public ArrayList<MetricsDesc> getResMetrics(CommunicationAddress resource, X509Certificate userCtx)` returns the list of metrics available on that resource.

4.5 Reservation Manager

ReservationManager is the class that oversees the creation and management of the reservations on the grid level. It is an entry point for JobMng and SAGA / XATI API, and it communicates with the node-level AllocationMngs.

The service is implemented in the `eu.xtreemos.xosd.reservationmanager.ReservationManager` class. However, the class is hosted by DIXI, so it should not be used directly.

Instead, the following interface should be used:

- `eu.xtreemos.xosd.services.SReservationManager` in other DIXI services, or
- `eu.xtreemos.xati.API.XReservationManager` in XATI client (the static methods of the class).

The interface exposes the following methods:

- `public String createEmptyReservation(X509Certificate userCertificate)`: Creates an empty reservation and returns its reservation ID.
- `public String createReservation(String query, X509Certificate userCertificate)`: Creates a reservation based on the JSDL query.
- `public String createReservationExplicit( ArrayList<ReservationRequest> requests, X509Certificate userCertificate)`: Creates a reservation based on the list of local allocation requests. If any of the requests fail, the whole reservation fails.
- `public Boolean updateReservation(String reservationId, String query, X509Certificate userCertificate)`: Updates the reservation by adding the local reservations as discovered according to the JSDL query. The reservation that is to be updated has to be empty.
- `public Boolean updateReservationExplicit(String reservationId, ArrayList<UpdateRequest> requests, X509Certificate userCertificate)`: Update previously created reservations.
- `public Boolean releaseReservation(String reservationId, X509Certificate userCertificate)`: Drop the reservation. The method releases all the local allocations, then removes the reservation from the directory. If any of the allocations cannot be released, it keeps them in the directory, as well as the reservation Id, so that they can be released later.

- `public Integer attachJob(String reservationId, String jobId, X509Certificate userCertificate)`: Attach a job to the reservation.
- `public Integer detachJob(String reservationId, String jobId, X509Certificate userCertificate)`: Detach the job from the reservation.
- `public String getReservationFromJob(String jobId, X509Certificate userCertificate)`: Query the reservation ID that a job is attached to.
- `public ArrayList<eu.xtreemos.xosd.reservationmanager.base.ReservationSlot> getReservationResources(String reservationId, X509Certificate userCertificate)`: Get the list of the resources and the related time constraints related to the reservation.
- `public FreeSlots getAllFreeSlotsFor(CommunicationAddress nodeAddress, String resourceID, X509Certificate userCert)`: Returns all free slots in a timetable for some metric on the node. The call performs the access rights check.
- `public FreeSlots getFreeSlotsFor(CommunicationAddress nodeAddress, String resourceID, GregorianCalendar from, GregorianCalendar to, X509Certificate userCert)`: Returns all free slots for a given resource and time frame on the node. The call performs the access rights check.
- `public ArrayList<String> getUserReservations(X509Certificate userCertificate)`: Retrieve the list of reservation IDs owned by the user.

4.6 Resource Allocator

- `public String createReservation(Request info)`: Creates a reservation with a given request. Returns new reservation id, if the reservation succeeds, otherwise, returns null. When reservation fails, coherent state is restored automatically.
- `public Boolean updateReservation(String reservationID, Request info)`: Updates a reservation with a request, but leaving the reservation id intact. Gets true, if all went well, false if it fails. The coherent state is restored automatically.
- `public Boolean updateReservations(ArrayList<UpdateRequest> requests)`: Creates a series of reservation updates. Note that if one fails, all other are nullified also and the state of all TimeTables is returned to the original state before the call.

- `public Boolean releaseReservation(String reservationID)`: Removes all entries regarding a reservation id. Note, that here the only criteria is a reservation id and not the whole needed info, for example how much of the resource needs to be freed as well. Use this method wisely, otherwise, please use create/update reservation methods and use remove request.
- `public ArrayList<String> getReservationResources(String reservationID)`: Returns all resources that are used by some reservation.
- `public Boolean attachToJob(String jobID, String reservationID)`: Associates a job with a reservation id. This is only on a semantic level, and should be used whenever reservations are dependent.
- `public Boolean detachFromJob(String jobID, String reservationID)`: Removes connection between a job and reservation id. This is managed by the user/ service and not automatically. Associations betweem reservations should be used whenever they are dependent.
- `public Boolean removeJob(String jobID)`: Remove any info regarding a job. Managed by the user/service.
- `public ArrayList<String> reservationsForJob(String jobID)`:
- `public FreeSlots getAllFreeSlotsFor(String resourceID)`: Returns all free slots in a timetable for some resource.
- `public FreeSlots getFreeSlotsFor(String resourceID, GregorianCalendar from, GregorianCalendar to)`: Returns all free slots for a given time frame for a specific resource.
- `public ArrayList<TTElm> selectAvailable(String resourceID, Integer amount)`: selects available from all elements (see `selectAvailableDT` description)
- `public ArrayList<TTElm> selectAvailableDT(String resourceID, Integer amount, GregorianCalendar from, GregorianCalendar to)`: selects all already reserved elements within a timetable for some resource (identified by resourceID), where the amount of available resource property CurrentAmount is smaller than MaxAmount by more than given parameter "amount". Also, the sharing value of such element must not be EXCLUSIVE.
- `public Integer initializeResource(String id)`: In order to make a new resource available and manageable (the ones that are not initialized in the startup process), one needs to register them, so that the system prepares a timetable for the reservations.

- `public ArrayList<String> getInitializedResources()`: returns all initialized resources
- `public Request createRequestsPurgeReservationsBefore(GregorianCalendar date)`: creates a request for purging all elements that finish before the given date. Since it is of no value to the executor and job manager to have reservations before current real time, all past reservations can be removed without any loss of info.
- `public Integer addResourceProperty(String resID, IResourceProperty property)`: dummy method that allows "on-hand" addition of properties. Should be replaced by proper initialization system ...
- `public ArrayList<TTElm> getSelection(String resID, Hashtable constraints)`: get all time table elements with selected attributes for a resource. The contraints are a map of attribute types and their values.
- `public ArrayList<ReservationSlot> getReservationsInfo(ArrayList<String> ids)`: For a list of the IDs obtained from the `createReservation(Request)`, the method builds a list of reservation slots descriptors, effectively returning the timetable entries usable for the reservations. Get required information for the ReservationManager about all reservations slots in the time table regarding some reservation id.
- `public ArrayList<String> createReservations(ArrayList<Request> reservations)`: creates a massive a sequence of reservations. returns the ids of reservatins with a 1:1 mapping regarding their place. If one of the reservations fails, all of them must fail also. If reservation fails, but restoring succeeds, the empty array is returned, otherwise, if the restoring fails also, the null object is returned.
- `public Boolean restoreForCheckpointBefore(ArrayList<String> reservationIDs)`: restores the state of all timetables just before the creation of the given reservation. If it was already restored or if there is no info about it, the call fails otherwise, it succeeds.

4.7 Resource Monitor

The Resource Monitor is the AEM's internal service that provides the interface to the readings from the node's Ganglia Monitoring daemon for the RCA client and the Resource Manager.

The full documentation is available as javadoc in /Support/Documentation/ in the WP3.3 SVN.

The class eu.xtreemos.xosd.resourcemonitor.ResourceMonitor exposes the following methods:

- `public Hashtable<String, String> queryResInfo()` queries the local monitor data provider for machine status and returns the values of the metrics being monitored on this node.
- `public Hashtable<String, Object> query()` queries the local monitor data provider for machine status, translates it into GLUE 1.2-compliant XML and transforms it into a Hashtable.
- `public ArrayList<ResourceDescriptorRecord> queryResourceDescriptor()` queries the local monitor data provider for machine status, translates it into GLUE 1.2-compliant XML and transforms it into a Hashtable.
- `public ArrayList<String> getResMetrics()` retrieves the list of currently supported and enabled metrics.
- `public Boolean addResAttribute(String attribute)` adds the attribute to the list of enabled metrics.

4.8 XtreemGCP Grid Checkpointing Service

The grid checkpointing service consists of a job level (CRJob Manager) and a job-unit level (CRExec Manager) component. Triggering a job checkpoint causes to interact with the job-level component only. The job checkpointing interface allows to checkpoint a single or dependent jobs each of them consisting of one or multiple job-units. A job checkpoint will be initiated by the following interface:

- `checkpointJobInit(String jobId, Integer resolveJobDependencies, String mode, String options, X509Certificate userCert)`

A checkpoint for a given jobId is issued. Dependencies between jobs can be resolved via the second parameter. The mode parameter is used to switch between checkpointing in the migration or in the fault tolerance context. The options parameter determines a desired checkpointing behaviour e.g. incremental/full checkpointing, etc. A given certificate allows only authorised actors to issue a checkpoint.
The job restart interface allows to restart a single or dependent jobs each of them consisting of one or multiple job-units. A job restart will be initiated by the following interface:

- `restartJobInit( String jobId, String checkpointVersion, Integer restartDependentJobs, ArrayList<String> destinationIP, ArrayList<String> destinationPort, String mode, X509Certificate userCert)`

A restart requires a jobId and a checkpoint image reference. One can switch between solely restarting one job, or a group of dependent jobs. The third and fourth parameter must be provided at least for each migrating job-unit to point out on which grid node it is to be restarted. The mode parameter determines whether to perform a restart in migration or fault tolerance context. A given certificate allows only authorised actors to issue a restart. Job migration is based on the sequence job checkpointing, job killing and job restart. A job migration will be initiated by the following interface:

- `migrateJob( String jobId, Integer resolveJobDependencies, String[] destinationVector, Integer options, X509Certificate userCert)`

The jobId references the job to be migrated. Job dependency resolvation can be activated by the second parameter. A formatted string defines which job-unit shall migrate to which destination node (string format: 'JobUnitID:IPAdress-JobUnitID:IpAdress...'). Using the options parameter a desired checkpointing behaviour can be determined. A given certificate allows only authorised actors to issue a migration.

4.9 XATI and C-XATI

AEM automatically generates XATI (JAVA) and C-XATI interfaces.

The best way to see the expanded interfaces documentation is the html javadoc-generated reference inside /Support/Documentation/ for the involved classes or the header files in case of C-XATI.

4.9.1 XATI class

The Java client code consists of the `eu.xtreemos.xati.XATI` class, and a set of generated classes for accessing the service calls. The main XATI class is statically present and used by all the generated classes. It offers some control of the way the client program communicates with the rest of the XtreemOS infrastructure. The following properties can be used:

- `public static CXATIConfig config`: the structure containing the current configuration used by the current and future service calls. The XATI reads the values during the static class initialisation. If the configuration file or the path to the configuration (`/.xos` by default) do not exist, it creates them, setting the default values. The following properties are available:
  - `CommunicationAddress address`: the address of the access point of the client. While the XATI does not serve service requests, but it needs to listen for responses of the requests issued by the client itself.
  - `CommunicationAddress xosdaddress`: the address of the daemon that the requests from the client will be sent to.

  - `String useSSL`: Defines whether the SSL will be used for the communication with the xosd. The node will be identified using the `certificateLocation` and `privateKeyLocation` for the public/ private key pair. The value can be changed during the runtime, but to actually make the changes have any effect, the **XATI.restart()** needs to be called.
  - `String certificateLocation`: This certificate is for XATI – XOSD communication and is used for the SSL handshake. It can either be a .crt file containing the public key of the communication service, or a full truststore containing the public, private key and the certification chain. The value can be changed during the runtime, but to actually make the changes have any effect, the **XATI.restart()** needs to be called.
  - `String userKeyFile`: This is where client key is stored. XATI does not use the option value, but the program using XATI can use it to load the key. It is up to the programmer using XATI to check for any changes of the value.
  - `String privateKeyLocation`: This is where the client's key is stored. If the **useSSL** is a .key certificate, then this path points to the corresponding private key. Otherwise the value is ignored. The value can be changed during the runtime, but to actually make the changes have any effect, the **XATI.restart()** needs to be called.
  - `String trustStoreSSL`: The path and the filename to the user certificate that can be used for authentication and authorisation when submitting a job or performing other tasks from clients to AEM. XATI does not use the option value, but the program using XATI can use it to load the certificate and pass it to the API calls. It is up to the programmer using XATI to check for any changes of the value.
  - `String userCertificateFile`: The string denoting the network adapter to be used for listening to connections at, e.g. eth0. If the value is omitted, then the first non-localhost and non-vmware adapter will be used.
  - `String networkInterface`: The name of the network interface to be used by XATI.
  - `String schemasLocation`: Path to the XSDs defined in XATI.

These are the static methods that can be used from XATI:

- `public static void restart()`: Calls `stop`, then starts a new set of communication threads and queues. The command therefore drops any pending transactions, and starts anew. If the `config` values have changed, they will be used when connecting to the new values.

- `public static void stop()`: Stop all communication queues and communication threads, ending all communication with the xosd.
- `public static X509Certificate getUserCertificate()`: returns the user certificate as defined in the XATI configuration file. The XATI reads the certificate during the static intialisation, so the instance returned is an in-memory copy.
- `public static PrivateKey getPrivateKey()`: returns the user's private key as defined in the XATI configuration file. The XATI reads the key during the static intialisation, prompting the user for the password, so the instance returned is an in-memory copy.

4.9.2 Client access interfaces

XATI classes and methods are in the package "eu.xtreemos.xati.API" included in xati.jar. The final class adds and "X" to the class name which methods are exported.

For example createJob method for JobMng is converted to: `static public String eu.xtreemos.xati.API.createJob(String __jsdlFile, Boolean __startJob, String __reservationID, X509Certificate __userCtx)`

- XCDAMng
- XCommon
- XCRExecMng
- XCronDaemon
- XDaemon
- XExecMng
- XJobDirectory
- XJobMng
- XRCAClient
- XRCAServer
- XResAllocator
- XResMng
- XReservationManager

- XResourceMonitor
- XSRDSMng
- XVOPS
- XXMLExtractor
- XXMLService
- XXOSQuotaLib

For monitoring output, there are helper classes that transform the XML into a Java class.

4.9.3 XML Helper (java)

The helper classes are inside the package eu.xtreemos.xosd.utilities.jobinfo contains JobInfoList class. This class is basically a wrapper for the output of monitor API. Once constructed with the XML output from getJobInfo, the user may query it for any job and metric obtaining a MetricValue object: There are three polymorphic versions of getMetricValue depending on the scope of the requested metric:

- getMetricValue(jobId, metricName)
- getMetricValue(jobId, resourceId, metricName)
- getMetricValue(jobId, resourceId, pid, metricName)

To make it easier to work with monitor API output, there are three useful methods that return the list of jobs, resources or pids present in the XML:

- getJobs()
- getJobResources(String)
- getResourcePids(String, CommunicationAddress)

4.9.4 C-XATI

C-XATI methods are exported to libXATICA, the signature of the methods exported is similar to Java methods but with the return value as reference.

As an example JobMng.createJob is included in XCJobMng.h and its signature is:

```
int createJob(char* __jsdlFile,char __startJob,char* __reservationID,char*
__userCtx, char** returnValue)
```

The int return value is an error code, where 0 means no error, and negative values are defined in xos_errno.h header file. There is also a function in libXATICA to get a textual description from the error codes:

`char *xos_strerror(int n)` Gets the predefined message for the error code.

Exceptions generated in the java XOSd are translated to this error codes automatically and attached message can be queried with another function:

`char *xos_exceptionMsg()` Gets the message present in the last exception.

There are also helper classes that transform the XML into a C struct, and ArrayList management utilities.

In the next pages, we will list the C enumerations and error codes for the different C functions.

4.9.5 Enumerations

```
enum TypeOfInfo {
  XOS_BASIC=2,
  XOS_JOB_DEFINITION=4,
  XOS_RESOURCES_ALLOCATED=8,
  XOS_RESOURCES_CONSUMED=16,
  XOS_USER_METRICS=32,
  XOS_NO_BUFFER=64,
  XOS_XTRACE=128,
  XOS_ENABLE=256
};

enum InfoLevel {
  XOS_JOB=1,
  XOS_PROCESS,
  XOS_KERNEL
};

enum DependenceDirection {
  XOS_UP = 0,
  XOS_DOWN
};

enum ControlOperations {
  XOS_SUSPENDJOB=19,
  XOS_RESUMEJOB=18,
  XOS_CANCELJOB=15
};
```

```
enum MetricType {
  XOS_MTINT,
  XOS_MTDOUBLE,
  XOS_MTCHAR,
  XOS_MTSTRING,
  XOS_MTTIME
};

enum MetricScope {
  XOS_MSJOB,
  XOS_MSRESOURCE,
  XOS_MSJOBUNIT,
  XOS_MSPROCESS,
};

enum MetricBufSize {
  XOS_MBSHORT,
  XOS_MBMEDIUM,
  XOS_MBLONG,
};
```

4.9.6 Error codes

```
  /* System errors */
#define _xos_syserr_end 1000
#define _xos_syserr_start 900
#define XOS_ECOMMURCV 999
#define XOS_EPARSERCV 998
#define XOS_ECOMMUSND 997
#define XOS_EPARSESND 996

  /* System exceptions */
#define XOS_EXGENERIC     1   /* Generic exception */
#define XOS_EXBADRES      2   /* Bad resource */
#define XOS_EXBADATTRCERT 3   /* Bad attribute certificate of the resource */
#define XOS_EXRESNOTREG   4   /* Resource not registered */
#define XOS_EXUNKNOWNDT   5   /* Unknown datatype */
#define XOS_EXUNKNOWNOID  6   /* Unknown OID */
#define XOS_EXBADUSERCERT 7   /* Bad user certificate*/
#define XOS_EXREVDIR      8   /* Reservation directory generic exception */
#define XOS_EXREVDIRALRE  9   /* Reservation directory: job already attached */
#define XOS_EXREVDIRNOEMP 10  /* Reservation directory: not empty reservation */
#define XOS_EXREV         11  /* Reservation generic exception */
#define XOS_EXREVNORES    12  /* Reservation: not enough resources */
```

```
#define XOS_EXREVNOSLOT   13  /* Reservation: no free slots available */
#define XOS_EXREVLOCALLOC 14  /* Reservation: local allocation error */
#define XOS_EXREVRESELECT 15  /* Reservation: resource selection error */
#define XOS_EXREVACCDENY  16  /* Reservation: access denied */
#define XOS_EXREVNOEXIST  17  /* Reservation: does not exist */
#define XOS_EXJOBNOEXIST  18  /* JobID doesn't exist */
#define XOS_EXINVCERT     19  /* Invalid certificate */
#define XOS_EXOPNOALLOW   20  /* Operation not allowed */
#define XOS_EXNOACCES     21  /* Permission denied */
#define XOS_EXBADMETRIC   22  /* Bad metric */
#define XOS_EXMON         23  /* Monitor generic exception */
#define XOS_EXREVFUTURE   24  /* Reservation starts in the future */
#define XOS_EXREVEXPIRE   25  /* Reservation expired */
#define XOS_EXNORESAV     26  /* No resource available */
#define XOS_EXNORESPOL    27  /* No resource matches policies */
#define XOS_EXRESNOIREV   28  /* Resource not in reservation */
#define XOS_EXBADSIG      29  /* Bad signature */
#define XOS_EXSVCNORUN    30  /* Service not running */
```

4.9.7 XML Helper (C)

The usage of the helper is the following:

```
  JobInfoList myJil;
   ArrayList jobs;

   myJil = XML2JobInfoList(jInfoXML);
   if (myJil == NULL){
     fprintf(stderr, "parse error\\n");
     return -1;
   }

   int errcode = getJobs(jil, \&jobs);
```

The different helper methods provided are the next ones, they manage ArrayList (free - new) of different types and convert between XML and structs:

ArrayList

ArrayList newArrayList();
void freeArrayList(ArrayList array);
int addToArrayList(ArrayList array, char *el);
char *getFromArrayList(ArrayList array, int elno);
char *ArrayList2XML(ArrayList array);

CommunicationAddress
int CommAddress2str(const CommunicationAddress ca, char **str);
int str2CommAddress(const char *str, CommunicationAddress *ca);
int equalsCommAddress(const CommunicationAddress ca1, const CommunicationAddress ca2);
void freeCommAddress(CommunicationAddress ca1);

MetricsDesc
MetricsDesc newMetricsDesc();
MetricsDesc XML2MetricsDesc(const char *xmlin);
char *MetricsDesc2XML(const MetricsDesc md);
void setMetricsDescName(MetricsDesc md, const char *name);
void setMetricsDescDescription(MetricsDesc md, const char *description);
void setMetricsDescScope(MetricsDesc md, enum MetricScope ms);
void setMetricsDescType(MetricsDesc md, enum MetricType mt);
void setMetricsDescBufferable(MetricsDesc md, int bufferable);
void setMetricsDescBufSize(MetricsDesc md, enum MetricBufSize mb);
void freeMetricsDesc(MetricsDesc md);

JobInfoList
JobInfoList XML2JobInfoList(const char *xmlin);
void freeJobInfoList(JobInfoList jil);
int getJobs(const JobInfoList jil, ArrayList *array);
int getJobResources(const JobInfoList jil, const char *jobId, ArrayListCA *array);
int getResourcePids(const JobInfoList jil, const char *jobId, const CommunicationAddress ca, ArrayListInt *array);

MetricValue
int getMetricValue(const JobInfoList jil, enum MetricScope scope, const char *jobId, const CommunicationAddress resourceId, const int pid, const char *metricName, MetricValue *mv);
void freeMetricValue(MetricValue mv);
char *getFirstMetricValue(const MetricValue mv);
char *getFromMetricValue(const MetricValue mv, int elno);
int getMetricValueLength(const MetricValue mv);

ArrayListCA
ArrayListCA newArrayListCA();
int addToArrayListCA(ArrayListCA array, CommunicationAddress el);

CommunicationAddress getFromArrayListCA(ArrayListCA array, int elno);
void freeArrayListCA(ArrayListCA array);

ArrayListInt
ArrayListInt newArrayListInt();
int addToArrayListInt(ArrayListInt array, int el);
int getFromArrayListInt(ArrayListInt array, int elno);
void freeArrayListInt(ArrayListInt array);

5 SRDS - Resource Discovery

5.1 RSS

The Resource Selection Service (RSS) provides a simple interface defined by `eu.xtreemos.rss.protocol.QueryForwarder`. This interface can be obtained by invoking a static method `start(String configFilename)` on the `eu.xtreemos.rss.XtreemRSS` class, which creates and starts a local RSS instance. The `QueryForwarder` interface contains the following two methods.

- `public void forwardJSDLQuery(String jsdl, RssReply rssReply);`
  This method submits a JSDL query to the RSS and registers an `RssReply` handler which asynchronously receives the results.
- `public void forwardJSDLQuery(String jsdl, String xacml, RssReply rssReply)`
  This method submits a JSDL query to the RSS together with an XACML policy filter that specifies additional (security-related) resource constraints. Again, the results are asynchronously passed to the `RssReply` handler.

The `RssReply` handler is a user-provided object that implements the `eu.xtreemos.rss.misc.RssReply` interface. This interface contains only one method.

- `public void setGlueResponse(String GlueResponse);`
  This method is invoked on the handler when a query completes and returns full results. The results are encoded in the GLUE standard.

5.2 ADS

The ADS module provides interfaces for two different services: Job Directory and Resource Discovery. These interfaces are exposed within XtreemOS through XOSD. The methods that actually access the DHTs are implemented in an asynchronous way.

The class **eu.xtreemos.ads.connection.dixi.SRDSMng** exposes the following methods to allow to managing information about jobs.

- `public int putJob(String jobID, CommunicationAddress contactPoint, String userID);`
  Insert the information regarding a job into the directory.
- `public int modifyJobContactPoint(String jobID, CommunicationAddress newContactPoint);`
  Change the contact point of a job already present in the directory.
- `public int setAttribute(String jobID, Attribute attribute);`
  Add an attribute to a job, or update the attribute value if it is already existing.
- `public int removeJob(String jobID);`
  Remove all information about a job from the job directory.
- `public long getJobContactPoint(String jobID);`
  Query the contact point of a job currently in the directory.
- `public long getJobsByAttribute(Attribute attribute);`
  Query the keys that match the attribute pair *name* and *value* present in the *attribute* parameter.
- `public long getJobAttribute(String JobId, String attributeName);`
  Invoke a query for the attribute value related to the given job.
- `public Integer removeJobAttribute(String jobID, String attrName)`
  Remove an attribute associated to a jobID. This method can be used only for not reversed attibute such UserID.

The same class **eu.xtreemos.ads.connection.dixi.SRDSMng** exposes also the following methods in order to manage the resources information.

- `public long resourceQuery(String jsdlDocument, String xacmlDocument, X509Certificate userXOSDCert);`
  Invoke a resource discovery process. The content of JSDL query and the policy are passed as String. Note: at present, ADS does not check against the policy. The method starts the query in the asynchronous way. In order to retrieve the results the method `getResourceQueryResults(long, ArrayList)` is used.
- `public int registerResource(CommunicationAddress hostAddress);`
  Requests that the host to be added to the directory service. The host

address contains the IP of the resource and the port number that the AEM/DIXI is listening at, and, additionally, an IP of the NAT gateway for NAT traversal. Note: ADS does not implement this method yet.

- `public int removeResource(CommunicationAddress hostAddress);`
  Request that the host is removed from the directory service, effectively no longer being discoverable. The host contains the IP of the resource and the port number that the AEM/DIXI is listening at, and, additionally, an IP of the NAT gateway for NAT traversal. Note: ADS does not implement this method yet.

The class **eu.xtreemos.ads.connection.dixi.SRDSMng** exposes also the following method in order to push informations concerning Vivaldi coordinates and in order to search IPs by passing Vivaldi coordinates and radius.

- `public Integer pushingVivaldiCoordinates(Double X, Double Y, InetAddress IP);`
  This method push Vivaldi coordinates concerning an IP into the DHT. The coordinates are expressed as Double and IP by using InetAddress type.

- `public Integer neighborhoodSearchVivaldiCoords(Double x, Double y, Double ray, int numberOfResult);`
  This method takes a couple of values x,y (coordinate vivaldi), a radius and the required number of answers and searches into the coordinates grid the IPs situated inside the area bounded by the circle with center x,y and radius passed as argument.

The ADS module provides RMI interfaces equivalent to the previous, used mainly for testing purposes.

The Job Directory service RMI interface exposes the following methods:

- `public void JDSaddNewJob(String jobId, String JobMng, String userId)`
  Add a new Job to the Directory service; it takes care of adding it to secondary indexes in order to perform reverse queries (i.e. by UserID)

- `public void JDSupdateJobAttribute(String jobId, String attrName, String attrValue)`
  Update the value of a given attribute for a given Job; the Job must exist in the DHT.

- `public void JDSaddJobAttribute(String jobId, String attrName, String attrValue)`
  Add a new pair (attribute, value) to a specified Job

- `public String JDSgetJobAttributeValue(String jobId, String attrName)`
  Retrieve the value of a give attribute for a specified Job; the JobID and its attribute shall exist.

- `public String[] JDSgetJobByAttributeValue( String attrName, String attrValue)`
  List the JobIDs of all those Jobs having a given (attribute, value); it needs to be able to do the reverse query (only possible on UserID, at the moment); it raises an exception otherwise.

- `public void JDSremoveJobAttribute(String jobId, String attrName)`
  Remove an attribute from an existing Job

- `public void JDSremoveJob(String jobId)`
  Delete any information concerning a given jobID from off the DHT

The Resource Discovery service RMI interface exposes the following method:

- `public String AEMlocate(String jsdlQuery)`
  Resolve a JSDL query even with dynamic attributes. Call the RSS module for retrieving informations about the static attributes.

The Vivaldi coordinates manager service RMI interface exposes the following method:

- `int pushingCoordinates(Double X, Double Y, InetAddress IP)`
  Push Vivaldi coordinates concerning IPs

- `ArrayList<String> resolveRange(Double x, Double y, Double ray, int numberOfResult)` Search IPs into the circle formed by x,y and radius ray.

6 XtreemFS

XtreemFS services (MRC, DIR and OSD) do not use other XtreemOS services. The communication protocol used by XtreemFS components can be found in Deliverable D3.4.5. Updated versions will be published in the svn in `trunk/docs`. As this is an internal protocol it may change at any time without further notice.

The XtreemFS client can optionally use the AMS service for account mappings (`amsclient_invmappinginfo_internal` and `amsclient_mappinginfo_internal`). There is also an optional plugin for the MRC that uses Java packages from `org.xtreemos.wp35` to read XtreemOS certificates.

7 OSS

The Object Sharing Services (OSS) includes all interfaces related to sharing of volatile memory objects. The interfaces described below are the internal interface that are all provided by the XOSAGA API. More information can be found in deliverables D3.4.4, D3.4.5 and D3.1.5.

- `int oss_startup(const char *addr, const char *listen_port, const char *bootstrap_addr, const char *bootstrap_port);`
- `void *oss_alloc(size_t size, oss_consistency_model_t consistency_model, oss_alloc_attributes_t *attributes);`
- `void oss_free(void *ptr);`
- `oss_transaction_id_t oss_bot(oss_transaction_priority_t priority, oss_transaction_attributes_t *attributes);`
- `int oss_eot(oss_transaction_id_t taid );`
- `int oss_abort(oss_transaction_id_t taid, oss_ta_abort_t *type );`
- `oss_permit_abort(oss_transaction_id_t taid );`

8 Communication

8.1 DIXI

DIXI (DIstributed eXtreemos Infrastructure) is a framework for hosting services, which includes facilities for communication between services on remote hosts, message redirection and secure communications. It does not have API as such. However, some functionality can be obtained by accessing service calls of a special service named `Daemon`. There is also a singleton class `Site` that enables access to certain information and additional functionality.

8.1.1 Daemon service

This service maintains the information needed by each daemon process (**xosd**) on the services running within the daemon, and the addresses of access points of other daemons in the network. In the static set-up, this service also provides the functionality of the service directory.

The interfaces to the service are as follows:

- `eu.xtreemos.xosd.services.SDaemon` in other DIXI services, or
- `eu.xtreemos.xati.API.XDaemon` in XATI client (the static methods of the class).

The interface exposes the following methods:

- `public java.util.ArrayList<CommunicationAddress> getDaemons()`: Retrieve a list of access point addresses of the known DIXI daemons.
- `public java.util.ArrayList<String> getServiceList()`: Retrieve the list of the services currently running on this daemon.
- `public Integer registerMyServices()`: Publish the services of this daemon to the service directory.
- `public java.util.ArrayList<CommunicationAddress> getNodesRunningService(String serviceName)`: List the access point addresses running the given service.
- `public Integer kill()`: Unregister this daemon and stop it.

8.1.2 Site class

This class exists as a static singleton whitin the xosd, and can be used by the services to gain information on the current runtime. It has the following properties that can be read:

- `CommunicationAddress address`: The address of the current daemon's access point. This is the address that accepts incoming service requests.
- `int xmlport`: the port number of the XML protocol, used, e.g., by C-XATI.
- `String xosdRootDirectory`: the path where the DIXI resides.
- `CommunicationAddress rootaddress`: the access point of the root daemon (xosd with service directory).
- `KeyStore trustedKeyStoreSSL`: the keystore containing the public certificates of the trusted CAs for the incoming SSL communications.

The class also has utility methods:

- `void netSend(CommunicationAddress address, ServiceMessage sm)`: sends the service message to the access point at the given address.

8.2 Pub/Sub

The PubSub module provides interfaces for the publish-subscribe service. The interface is currently exposed within XtreemOS as a Java library.

Publishing topics

```
String topic;
String content;
OtpErlangString otpTopic;
OtpErlangString otpContent;

Scalaris sc = new Scalaris();
sc.publish(topic, content);       // publish(String, String)
sc.publish(otpTopic, otpContent); // publish(OtpErlangString,
                                  //         OtpErlangString)
```

For the full example, see de.zib.scalaris.examples.ScalarisPublishExample

Subscribing to topics When an item is published under the registered topic, an HTTP Request will be made to the registered URL. The body of the request will contain a small JSON document with the contents of the published item.

```
String topic;
String URL;
OtpErlangString otpTopic;
OtpErlangString otpURL;

Scalaris sc = new Scalaris();
sc.subscribe(topic, URL);       // subscribe(String, String)
sc.subscribe(otpTopic, otpURL); // subscribe(OtpErlangString,
                                //           OtpErlangString)
```

For the full example, see de.zib.scalaris.examples.ScalarisSubscribeExample

Unsubscribing from topics Unsubscribing from topics works like subscribing to topics with the exception of a NotFoundException being thrown if either the topic does not exist or the URL is not subscribed to the topic.

```
String topic;
String URL;
OtpErlangString otpTopic;
OtpErlangString otpURL;
```

```
Scalaris sc = new Scalaris();
sc.unsubscribe(topic, URL);       // unsubscribe(String, String)
sc.unsubscribe(otpTopic, otpURL); // unsubscribe(OtpErlangString,
                                  //             OtpErlangString)
```

Getting a list of subscribers to a topic

```
String topic;
OtpErlangString otpTopic;

Vector<String> subscribers;
OtpErlangList otpSubscribers;

// non-static:
Scalaris sc = new Scalaris();
subscribers = sc.getSubscribers(topic);
// getSubscribers(String)
otpSubscribers = sc.singleGetSubscribers(otpTopic);
// getSubscribers(OtpErlangString)
```

9 Virtual Nodes

This section describes the use of the Virtual Nodes library. It is structured in accordance to the different use cases one might find: developing an application using Virtual Nodes' Java-RMI-like user interface; configuring a Virtual Nodes instance; and finally, implementing a custom middleware layer. To clarify things, we develop a simple dictonary application which we use to explain the usage of the API.

9.1 Java RMI-like Application Programmer API

Virtual Nodes comes with an API that is adapted from Java's RMI layer. In this section we present this API and show how to use it.

Implementing a Replicated Service

A service which is to be run replicated using Virtual Nodes has to implement the interface `rmi.RRMIObject`[5]. Furthermore, it has to implement `java.io.Serializable`. This is due to the fact that for starting new replicas, the replica state has to be transferred. In consequence the serialisation

[5]all package names in this document that do not start with `java.` are relative to `eu.xtreemos.vnode`.

methods `readObject` and `writeObject` have to be implemented accordingly where neccessary. As an optional step methods can be marked to be read-only using the `rmi.Readonly` annotation.

For our example this looks as follows.

```
public class DictionaryService implements RRMIObject, Serializable, Dictionary {

        private final Hashmap<String,String> entries = new Hashmap<String,String>();

        @Override @Readonly
        public String getEntry(String key) throws NoSuchObjectException {
         String val = entries.get(key);
         if(val == null) throw new NoSuchObjectException();
         return val;
        }

        @Override
        public void addEntry(String key, String value){
         entries.put(key, value);
        }
}
```

The Virtual Nodes Registry

In ordernary Java RMI applications one uses the Java registry `rmiregistry` or the classes provided by `java.rmi.registry`. As this registry has some limitations such as that it is not possible to insert entities from a remote host, Virtual Nodes come with a wrapper that is located in the `rmi.registry` package and can be accessed by `rmi.registry.RegistryAccessor`. This registry is not fault-tolerant and constitutes a single point of failures. Thus, it should not be relied upon. Instead we recommend the use of the XtreemOS directory service as a persistent reference.

Starting a Replicated Service

Starting a new replica comprehends the following steps.

- Create an instance of the service to be replicated
- Export the instance using `rmi.Exporter`
- Add the proxy to the registry

In our example this translates to the following lines of code.

```
    public static Dictionary startService() throws ExportException,
                                    RemoteException, AlreadyBoundException {
            DictionaryService dicts = new DictionaryService();
            Remote proxy = (Remote) Exporter.exportObject(dicts).proxy;

            Registry reg = RegistryAccessor.INSTANCE.findRegistry();
            reg.bind(RegistryAccessor.name, proxy);
            return (Dictionary) proxy;
    }
```

Starting a Client

At client-side, the only steps to be taken are to read the reference from the registry and to cast it to the respective type.

```
    public static Dictionary getClient() throws AccessException,
                                    RemoteException, NotBoundException {
            Registry reg = RegistryAccessor.INSTANCE.findRegistry();
            Dictionary dict = (Dictionary) reg.lookup(RegistryAccessor.name);

            return dict;
    }
```

Starting a New Replica

Starting a new replica is similar to acting as a client. In addition to getting the reference, it is required to cast it to `client.AdminMethods`, Virtual Nodes' administration interface, and then invoke `startNewReplica(MiddlewareAdapter adapter)`. `adapter` may be `null` just as shown in the example code below. In case it is non-null, the adapter will be invoked each time the group of replicas changes. This mechanism can be used to update the information in the registry.

```
public static void startReplica() throws AccessException,
                                    RemoteException, NotBoundException {
        Dictionary dict = getClient();
        AdminMethods adm = (AdminMethods) dict;
        adm.startNewReplica(null);
}
```

9.2 Configuration Parameters

Virtual Nodes can be configured using a configuration file. The location of this file is set using the `-D` switch with `vnode.config.file=<filelocation>`. If the value is not set, the framework searches a file called `vnode.config` in the ∼`/.vnode/` and in `/etc/xos/vnode/`. Besides the use of a configuration file, all configuration parameters can be set directly using the `-D` switch. Parameters set this way always overrule the values of the configuration file. If neither a configuration file can be found or the configuration file does not contain a value for a parameter nor the parameter is set via `-D`, Virtual Nodes will use a default value for that parameter. In the following we present a list of all configuration options. We distinguish *global (g)* options which can only be set for the first replica and will be silently ignored for all others, and *local (l)* options which can be set for each replica and client.

- `vnode.config.file`: the path where the configuration file is located. If it is not set, the framework looks up the configuration in ∼`/.vnode/vnode.config` and then in `/etc/xos/vnode/vnode.config`. Values set as system property will overrule those set in the configuration file. Setting this value in the configuration file will has no impact.
- `vnode.config.groupcom`: determines the group communication to be used: takes one of the values {JGROUPS, DUMMY} *(g)*
  - `JGROUPS` provides the following options:
    - `vnode.config.groupcom.jgroups.portrange`: the portrange JGroups is using *(l)*
    - `vnode.config.groupcom.jgroups.port`: the local port Jgroups tries first *(l)*
    - `vnode.config.groupcom.jgroups.fdtimeout`: the timeout of the failure detector *(l)*
    - `vnode.config.groupcom.jgroups.address`: the local IP address to bind to *(l)*
    - `vnode.config.groupcom.jgroups.groupname`: the groupname to be used *(g)*
- `vnode.config.replication`: determines the replication protocol: takes one of the values {ACTIVE} *(g)*
- `vnode.config.scheduler`: determines which scheduling algorithm is used: takes one of the values {MAT, SEQ}, might be overridden by the replication protocol *(g)*
- `vnode.config.contact`: determines the external communication protocol: takes one of the values {TCP, DUMMY} *(g)*

  - Options for `TCP`
    * `vnode.config.contact.tcp.address:` determines which local address to bind to. If not set, the server binds to all local addresses *(l)*
    * `vnode.config.contact.tcp.port:` determines which port to use. if not set, the port will be chosen by random *(l)*

- `vnode.config.client.semantics:` determines which invocation semantics the client requires: takes one of the values {BEST_EFFORT, AT_MOST_ONCE} *(l)*
- `vnode.config.client.quantity:` determines how many reply the client requires: takes a positive integer value {1, ... } *(l)*
- `vnode.config.client.selector:` classname of the class used for selecting the next replica to contact to *(l)*
- `vnode.config.client.checker:` classname of the class used for selecting the reply *(l)*

`rmi.registry.RegistryAccessor` uses the following properties:

- `vnode.rmi.registry.accessor.name:` the name the stub is bound to
- `vnode.rmi.registry.accessor.host:` the host the registry resides on
- `vnode.rmi.registry.accessor.port:` the port used by the registry

A list of system properties that are reserved for internal or future use and must not be set by users:

- `vnode.config.done`
- `vnode.config.groupcom.jgroups.knownhosts:` the hosts known to the Jgroups system (persistent property)
- `vnode.config.groupsize:` the number of replicas to be used: takes a positive integer value {1,...}

9.3 Implementing a Middleware Layer

The way Virtual Nodes are designed allows for multiple middleware layers on top of the replication logic. A middleware layer is responsible for parameter (un)marshalling. In addition, it dispatches methods at server-side and provides information about the method character, that is, whether an invokation is read-only or not. The layer has to implement three classes

- a dispatcher class that implements `execution.ReplicatedObject`
- a serialiser implementing `ReplicatedObjectSerialiser` that is able to serialise the `execution.ReplicatedObject`
- a method identifier extending `common.MethodId`

```
public interface ReplicatedObjectSerialiser {
        public void serialise(ReplicatedObject ro, OutputStream out) throws IOException;
        public ReplicatedObject deserialise(InputStream in) throws IOException;
}

public interface ReplicatedObject {

        public boolean isReadOnly(MethodId method)
                                        throws UnknownMethodException;

        public SingleReply dispatch(ReplicaId id, MethodId method, byte[] arguments)
                                        throws UnknownMethodException;
}
```

The middleware layer-specific `MethodId` is created at client-side by the middleware layer and serialised afterwards by the core. At server-side the core is not aware which middleware layer is being used and thus, cannot deserialise the `MethodId` correctly. Instead, it creates an instance of `common.UnresolvedMethodId` containing the required deserialisation information. As a consequence the instances of `common.MethodId` that being passed to `isReadOnly()` and `dispatch()` may also be of type `UnresolvedMethodId`. The implementation has to check this and transform it to the correct type if necessary. In order to avoid multiple transformations `UnresolvedMethodId.setMethodId()` allows to set the resolved `MethodId`. The core will use this one as soon as the value has been set.

At client-side the middleware layer uses instances of `client.ClientBase` in order to provide the necessary funtionality.

```
public class ClientBase {
        public SingleReply sendCall(MethodId methodId, byte[] parameters)
                                                throws ConnectException;

        public AdminMethods generateAdminMethodsInstance(ReplicatedObjectSerialiser ser);
}
```

`sendCall` is used to invocate remote methods provided by the service itself. Administration methods are handeled by an instance of `AdminMethods` which is cretated by `generateAdminMethodsInstance` in `ClientBase`. All invocations to an administration method have to be relied to this object. In consequence the middleware layer has to ensure that a service does not offer methods that conflict with administration methods. How this is done is left open to the implementor.

10 Interfaces Specific to the Cluster Flavour

In this section, we only mention the interfaces specific to the cluster flavour.

10.1 LinuxSSI

LinuxSSI supports the POSIX API like the vanilla Linux kernel. It also supports the POSIX thread API. In the next sections we describe the LinuxSSI-specific API.

10.1.1 Capabilities

- `int krg_capset (krg_cap_t * new_caps)` : Set capabilities `new_caps` for the current process.
- `int krg_capget (krg_cap_t * old_caps)` : Get capabilities for the current process into `old_caps`.
- `int krg_pid_capset (pid_t pid, krg_cap_t * new_caps)` : Set capabilities `new_caps` for a given process.
- `int krg_pid_capget (pid_t pid, krg_cap_t * old_caps)` : Get capabilities for a given process into `old_caps`.
- `int krg_cap_geteffective (krg_cap_t * cap)` : Return the effective capabilities of the capability structure `cap`.
- `int krg_cap_getpermitted (krg_cap_t * cap)` : Return the permitted capabilities of the capability structure `cap`.
- `int krg_cap_getinheritable_permitted (krg_cap_t * cap)` : Return the permitted inheritable capabilities of the capability structure `cap`.
- `int krg_cap_getinheritable_effective (krg_cap_t * cap)` : Return the effective inheritable capabilities of the capability structure `cap`.

10.1.2 Hotplug

- `int krg_get_max_nodes(void)`: Get the maximum number of nodes in the cluster.
- `int krg_get_max_clusters(void)`: Get the maximum number of subclusters in the cluster. In the current version of LinuxSSI, only one subcluster is supported.
- `void krg_clear_node_set(struct krg_node_set *item)` : Initialize to zero the krg_node_set `item`.
- `void krg_node_set_add(struct krg_node_set *item, int n)` : Enable a node in the krg_node_set structure `item`.
- `int krg_nodes_add(struct krg_node_set *node_set)`: Add enabled nodes in the krg_node_set structure to the cluster.
- `int krg_nodes_remove(struct krg_node_set *node_set)`: Remove enabled nodes in the krg_node_set structure to the cluster.
- `int krg_nodes_fail(struct krg_node_set *node_set)`: Declare as failed enabled nodes in the krg_node_set structure to the cluster.
- `int krg_nodes_poweroff(struct krg_node_set *node_set)`: Power off enabled nodes in the krg_node_set structure to the cluster.
- `struct krg_nodes* krg_nodes_status(void)`: Get status of all nodes in the cluster.
- `struct krg_clusters* krg_cluster_status(void)`: Get status of all subclusters. In the current version of LinuxSSI, only one subcluster is supported.
- `int krg_cluster_start(struct krg_node_set *krg_node_set)`: Start enabled nodes in the krg_node_set structure to the cluster.
- `int krg_cluster_shutdown(int subclusterid)`: Shutdown the cluster identified by `subclusterid`. In the current version of LinuxSSI, only the subclusterid 0 is supported.
- `int krg_cluster_reboot(int subclusterid)`: Reboot the cluster identified by `subclusterid`. In the current version of LinuxSSI, only the subclusterid 0 is supported.

10.1.3 Process management

- `int get_cpu_id(void):` Return the node id of the local machine (will be available as get_node_id in the next versions of LinuxSSI).
- `int get_nr_cpu(void):` Return the number of nodes in the cluster.
- `int migrate (pid_t pid, int destination_node) :` Migrate the process `pid` to the node `destination_node`.
- `int migrate_self (int destination_node):` Migrate the current process to the node `destination_node`.
- `checkpoint_infos_t application_checkpoint_from_appid (media_t media, long app_id, int signal):` Checkpoint the application identified by `app_id` to the media `media` (disk or memory) and send the signal `signal` at the end of the checkpoint.
- `checkpoint_infos_t application_checkpoint_from_pid(media_t media, pid_t pid, int signal):` Checkpoint the application in which the process `pid` is involved to the media `media` (disk or memory) and send the signal `signal` at the end of the checkpoint.
- `int application_restart (media_t media, long app_id, int chkpt_sn, int flags):` Restart the application identified by `app_id` from the media `media` (disk or memory).

10.1.4 kDFS Kernel Distributed File System

Being integrated in the Linux VFS, kDFS supports the standard POSIX file system interface. kDFS has been checked for POSIX compliance with the POSIX File System Test Suite[6] version 20080412.

10.1.5 Pluggable Probes and Scheduling Policies Framework (PlugProPol)

Pluggable Probes and Scheduling Policies Framework (PlugProPol) is an infras- tructure which enables user to write his own probes and scheduling policies and add them to the system in runtime (without the need to restart the whole cluster). If a user wants, for example, to monitor disk usage on his local machine, he only implements a proper probe and plugs it to PlugProPol in runtime. This makes the scheduling much more configurable since no reboot is needed.

All the probes and scheduling policies are implemented as Linux kernel mod- ules, they run in kernel space and are able to access kernel data structures directly.

[6]http://www.ntfs-3g.org/pjd-fstest.html

The developers that would like to implement probes and scheduling policies can find all the information about the PlugProPol framework in [3].

10.2 DRMAA

The best reference documentation for developing DRMAA applications is certainly the DRMAA specification [1] itself, and the DRMAA C language binding [2]. They contain detailed descriptions of all the important DRMAA functions for job submission and control. A summarized description of those functions can also be found in the **"drmaa.h"** which is located in the "lib" subdirectory of the LinuxSSI DRMAA source tree:

- **int drmaa_init(const char* contact, char* error_diagnosis, size_t error_diag_len)**: initializes DRMAA library and creates new DRMAA sesssion. This function must be called before any other DRMAA function.
- **int drmaa_exit(char* error_diagnosis, size_t error_diag_len)**: disengages from DRMAA library and allows DRMAA library to perform any necessary internal cleanup. It ends current DRMAA session, but doesn't affect any jobs (e.g. queued and running jobs will remain queued running).
- **int drmaa_run_job(char *job_id, size_t job_id_len, const drmaa_job_template_t *jt, char *error_diagnosis, size_t error_diag_len)**: submits a single job with the attributes defined in the jt job template. Upon success, up to job_id_len characters of the submitted job's job identifier are stored in the job_id buffer.
- **int drmaa_run_bulk_jobs(drmaa_job_ids_t **job_ids, const drmaa_job_template_t *jt, int start, int end, int incr, char *error_diagnosis, size_t error_diag_len)**: submits a set of parametric jobs which can be run concurrently. The attributes defined in jt job template are used for every parametric job in the set. Each job in the set is identical except for its index. The smallest valid value for start is 1. The largest valid value for end is $2^{31} - 1$. The start value must be less than or equal to the end value and only positive index numbers are allowed. On success, an opaque job id string vector containing job identifiers for all submitted jobs is returned into job_ids.
- **int drmaa_job_ps(const char *job_id, int *remote_ps, char *error_diagnosis, size_t error_diag_len)**: retrieves the program status of the job identified by "job_id". The possible values of a program's status are defined in "drmaa.h" (look for all the macros with the DRMAA_PS_* prefix).

- **int drmaa_control(const char *job_id, int action, char *error_diagnosis, size_t error_diag_len)**: enacts the action indicated by "action" parameter on the job identified by "job_id". The possible values of "action" parameter are defined in "drmaa.h" (look for all the macros with the DRMAA_CONTROL_* prefix).

- **int drmaa_wait(const char *job_id, char *job_id_out, size_t job_id_out_len, int *stat, long timeout, drmaa_attr_values_t **rusage, char *error_diagnosis, size_t error_diag_len)**: waits for a job identified by job_id to finish execution or fail. If the special string, JOB_IDS_SESSION_ANY, is provided as the job_id, this function will wait for any job from the session to finish execution or fail. In this case, any job for which exit status information is available will satisfy the requirement, including jobs which preivously finished but have never been the subject of a drmaa_wait() call. This routine is modeled on the "wait3" POSIX routine.

- **int drmaa_synchronize(const char *job_ids[], signed long timeout, int dispose, char *error_diagnosis, size_t error_diag_len)**: the drmaa_synchronize function causes the calling thread to block until all jobs specified by job_ids have finished execution. If job_ids contains DRMAA_JOB_IDS_SESSION_ALL, then this function waits for all jobs submitted during this DRMAA session as of the point in time when drmaa_synchronize() is called.

To demonstrate the use of DRMAA library, there are examples available in the "tests" subdirectory of the LinuxSSI DRMAA source tree. Below, you will find a list of examples along with the explanation on what they do:

- test-drmaa-init.c: demonstrates how to initialize a new DRMAA session and later terminate it,

- test-drmaa-job-template.c: demonstrates how to create a new job template which contains all the necessary information about the job the user wants to submit,

- test-drmaa-job-run.c: demonstrates how to submit job via the DRMAA interface,

- test-drmaa-bulk-jobs-run.c: demonstrates how to submit multiple instances of the same job (i.e. a batch job) by invoking a single DRMAA function. Each job instance invokes the same command, however the parameters can be different,

- test-drmaa-job-ps.c: demonstrates how to retrieve the status of a job that was already submitted to the DRM system,

- test-drmaa-job-wait.c: demonstrates how to use DRMAA interface to wait for the completion of a specific job,
- test-drmaa-job-synchronize.c: demonstrates how to use DRMAA interface to wait for the completion of multiple jobs that were submitted to the DRM system,
- test-drmaa-file-streams.c: demonstrates how to redirect the input to be read from a file instead of stdin and the output to be written to a different file instead to stdout.

11 Interfaces Specific to the Mobile Device Flavour

In this section, we only mention the interfaces specific to the mobile device flavour.

11.1 libxos_getcred

This API just includes one function:

```
char * xos_getcred(char *configuration_name);
```

This function returns a credential in PEM format. The API uses `libcredstore` to implement single sign-on: if the credential is stored in the credstore, it is automatically retrieved, but if the credstore is empty or the credential is labelled with other configuration name different from the `configuration_name` parameter, `startxtreemos` (actually, /usr/bin/runcredagent that in XtreemOS is a symbolic link to startxtreemos) is launched (with `configuration_name` as the paremeter, if not NULL) to obtain a credential that is then saved in the credstore.

`xos_getcread` reads the configuration file `/etc/xos/configname_alias` to convert the configuration name from any of its aliases, as defined in that file.

The `configuration_name` parameter can be NULL. In this case, if the credstore is not empty, the credential is accepted without checking the configuration name registered. If the credstore is empty, a new credential is stored with configuration name "default".

In order to compile an application using this library, use the `-lxos_getcred` compilation option. The source code must include `xos_getcred.h`.

11.2 libxos-credagent

`libxos-credagent` is a library to implement a pluggable, modular system to get credentials. The objective of the library is to allow administrators to change the method used by applications to get the credentials to authenticate in a Single Sign-On (SSO) system, without source code modification. The basic interface for this library is:

```
char *xos_credagent_getcred(char *configuration);
```

This function returns the credential using the credagent configuration specified in file `/etc/xos/creds/<configuration>.conf`. The function returns NULL if there is any error (e.g. the user is not authorized to read the credential). The returned credential must be `free`'d by the caller.

Additionally, and in order to read any additional parameters from these configuration files, applications have the following interface available:

```
int xos_setconfigenv(char *config_name,char *section);
```

This functions maps all the parameters of the specified section in environment variables, which can be read using `getenv`. The function returns zero on success.

However, it is recommended that end-user applications do **not** invoke `libxos-credagent` directly, but run a wrapper such as `startxtreemos` instead. This kind of application has permission to read the `/etc/xos/creds` directory.

The advised behavior of applications is to try to obtain the credential from a cache (a credential store, e.g. using libcredstore, see D2.3.4 and then, only if it is not available, to run `startxtreemos` (which in turn invokes `libxos-credagent`). This behaviour is implemented in `libxos-getcred`.

Here is a summarized description of xos_credagent.h functions:

- **char* xos_credagent_getcred (char* configuration)**: get credential invoking the credagent module associated to configuration_name to get it.
- **void xos_credagentso_destroy (CREDAGENT_HANDLE handle)**: function to freed the handle obtained with xos_credagentso_instance when not needed anymore.
- **char* xos_credagentso_getparameter (CREDAGENT_HANDLE handle, char* key)**: get a parameter value from "credagent" section in configuration file.
- **CREDAGENT_HANDLE xos_credagentso_instance (char* config_name, char** name)**: function to obtain a handle needed to use libxos-credagent API for credagent modules.
- **int xos_creduiagent_ask_code (CREDUIAGENT_HANDLE handle, char* message, char** code, int max_length, char retry)**: Use creduiagent module to ask a text to user. This text is not asked as a password: the user may see what he/she type.
- **int xos_creduiagent_ask_confirmation (CREDUIAGENT_HANDLE handle, char* message)**: Ask confirmation to user.
- **int xos_creduiagent_ask_login_password (CREDUIAGENT_HANDLE handle, char* message, char** login, char** password, int max_length, char retry)**: Use creduiagent module to ask user a login and password or a password only.
- **int xos_creduiagent_ask_login_pin (CREDUIAGENT_HANDLE handle, char* message, char** login, char** pin, int length, char retry)**: Use creduiagent module to ask user a PIN number and (optionally) a username.

- **void xos_creduiagent_destroy (CREDUIAGENT_HANDLE handle)**: function to freed the handle obtained with xos_creduiagentso_instance when not needed anymore.
- **void* xos_creduiagent_get_func (CREDUIAGENT_HANDLE handle, char* name)**: get a function pointer to a method implemented in creduiagent module.
- **char* xos_creduiagent_getparameter (CREDUIAGENT_HANDLE handle, char* key)**: get a parameter value from "creduiagent" section in configuration file.
- **CREDUIAGENT_HANDLE xos_creduiagent_instance (char* config_name, char** name)**: function to obtain a handle needed to use libxos-credagent API for interaction with the creduiagent module specified in the credential configuration file.
- **int xos_creduiagent_show_error (CREDUIAGENT_HANDLE handle, char* message)**: Show a error message to user using the specified creduiagent.
- **int xos_setconfigenv (char* config_name, char* section)**: utility function to set configuration parameters from a section in environment variables.

11.3 libcdaclient

`libcdaclient` is a C library developed in order to get a credential from a CDA server or from a CDAProxy. It is used in several applications such as `cdacclient`, `cdaproxy` and in the `xos_credagent_cdaclient.so` module. A program that uses `libcdaclient` must include the header `cdaclient.h` and be linked with the `-lcdaclient` option.

The ordinary method to obtain a credential from a CDA server is:

```
int cda_client(char *hostname,
              int port,char *CDAcertfilename,
              char *RootCAcertfilename, char *username,
              char *password, char *voname,char **credential,
              char **certificate);
```

Where the parameters are:

- The `hostname` and `port` of the CDA server (mandatory).
- `CDAcertfilename` (optional): if not `NULL`, it should contain the name of a file containing the server certificate of the CDA server in PEM format, to test that the SSL connection is really with CDA server and not with a rogue server (to prevent a man-in-the-middle attack).

- `RootCAcertfilename` (optional): if not `NULL`, it should be the name of a file containing the root CA certificate (in PEM format) that signs the obtained credential. This is used to verify that the credential is authentic.
- `username` and `password` (mandatory): these are used to authenticate in the CDA server, and in order to indicate the user for whom the credential will be generated.
- `voname` (mandatory): the VO to which the user belongs (in this credential, at least).
- `credential` (mandatory): a pointer to a variable that will be filled with the credential in PEM format. If the `certificate` parameter is also filled in, only the private key will be stored in `credential`. If `certificate` is `NULL`, the certificate will also be included in `credential`.
- `certificate` (optional): the certificate part of the credential. This parameter may be `NULL`, and in that case `credential` will store the private key and the certificate.

The function returns zero on success.
A variant of this function is:

```
int cda_client_defergen(char *hostname,
              int port,char *CDAcertfilename,
              char *rootCAcertfilename, char *username,
              char *password, char *voname,char **credential,
              char **certificate);
```

This function is a variant of previous one. The only difference is that the RSA private key is generated after connecting and authenticating. This variant is useful for the implementation of CDAProxy, where the RSA key is generated by the proxy and the proxy does not authenticate the user but the CDA server. This is done to avoid denial of service attacks.

```
int cda_client_nossl(char *hostname,int port,
                       char *username,char *password,
                       char *voname, char **credential,
                       char **certificate);
```

This function is another variant of the first function, that uses a plain connection with the CDA server instead of a TLS/SSL one. Currently this function is not very useful, since the CDA server only accepts TLS/SSL connections, but could be useful in special cases where the connection with the CDA is trusted.

The `libcdaclient` library also supports **obtaining credentials through a CDAProxy**. In this case, the interface is slightly different:

```
int cda_client_proxy(char *hostname,
              int port,char *Proxycertfilename,
              char *RootCAcertfilename, char *username,
              char *password, char *voname,char **credential,
              char **certificate);
```

The parameters of this function are similar to those in `cda_client()`, but in this case the the `host` and `port` are those of the proxy, not the CDA ones. This function returns zero on success.

A variant of this function, for use when connecting to the CDAProxy through a plain connection (i.e. without TLS/SSL), is also available:

```
int cda_client_proxy_nossl(char *hostname,int port,
                        char *username, char *password,
                        char *voname, char **credential,
                        char **certificate);
```

11.4 libwrapopen

This library does not have a specific API, since transparency is its main objective. In order to use it, an environment variable must be set:

```
export LD_PRELOAD=/usr/lib/libxos_wrapopen.so
```

Once this is done, applications which are not SUID nor SGID will be able to read the credential corresponding to a configuration name whenever they open the file `::xos:configname` or `/::xos:configname`.

Additionally, libwrapopen can be further automatized to load the credential specified in the environment variable `XOS_WRAPOPEN_CONFIGNAME` when a directory listed in environment variable `XOS_WRAPOPEN_DIRS` (a comma separated list) is opened. This feature can be very useful for automounting XtreemFS volumes.

If the application only needs the private key part of the credential then it should open the following file `::xos::key:configname`. However, if the application only needs the certificate part of the credential then it should open the following file `::xos::cert:configname`.

References

[1] DRMAA 1.0 grid recommendation (GFD.133). `http://www.ogf.org/documents/GFD.133.pdf`, 2008.

[2] DRMAA C binding v1.0. `https://forge.gridforum.org/sf/docman/do/downloadDocument/projects.drmaa-wg/docman.root.ggf_13/doc5545`, 2008.

[3] XtreemOS consortium. Design and implementation of a customizable scheduler. Deliverable D2.2.6, November 2007.

[4] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Andre Merzky, John Shalf, and Christopher Smith. A Simple API for Grid Applications (SAGA). Grid Forum Document GFD.90, 2007. Open Grid Forum (OGF).