Public Key Infrastructure Specification

September 2002
Version 1.0
formal/02-09-04

An Adopted Specification of the Object Management Group, Inc.

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industry's premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at http://www.opengroup.org/ .

Intended Audience

The specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for object services; the benefits of compliance are outlined in the following section, "Need for Object Services."

Need for Object Services

To understand how Object Services benefit all computer vendors and users, it is helpful to understand their context within OMG's vision of object management. The key to understanding the structure of the architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification.*

- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.

The Object Request Broker, then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone cannot enable interoperability at the application semantic level. An ORB is like a telephone exchange: it provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between subscribers. Meaningful, productive communication depends on additional interfaces, protocols, and policies that are agreed upon outside the telephone system, such as telephones, modems and directory services. This is equivalent to the role of Object Services.

What Is an Object Service Specification?

A specification of an Object Service usually consists of a set of interfaces and a description of the service's behavior. The syntax used to specify the interfaces is the OMG Interface Definition Language (OMG IDL). The semantics that specify a services's behavior are, in general, expressed in terms of the OMG Object Model. The OMG Object Model is based on objects, operations, types, and subtyping. It provides a standard, commonly understood set of terms with which to describe a service's behavior.

(For detailed information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide).*

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- CORBA Platform Technologies
  - *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
  - *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
  - *CORBA Services*, a collection of specifications for OMG's Object Services. See the individual service specifications.
  - *CORBA Facilities*, a collection of specifications for OMG's Common Facilities. See the individual facility specifications.

- CORBA Domain Technologies
  - *CORBA Manufacturing*, a collection of specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
  - *CORBA Healthcare*, a collection of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
  - *CORBA Finance*, a collection of specifications that target a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
  - *CORBA Telecoms*, a collection of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

You may contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Service Design Principles

Build on CORBA Concepts

The design of each Object Service uses and builds on CORBA concepts:

- Separation of interface and implementation
- Object references are typed by interfaces
- Clients depend on interfaces, not implementations
- Use of multiple inheritance of interfaces
- Use of subtyping to extend, evolve and specialize functionality

Other related principles that the designs adhere to include:

- Assume good ORB and Object Services implementations. Specifically, it is assumed that CORBA-compliant ORB implementations are being built that support efficient local and remote access to “fine-grain” objects and have performance characteristics that place no major barriers to the pervasive use of distributed objects for virtually all service and application elements.
- Do not build non-type properties into interfaces

A discussion and rationale for the design of object services was included in the HP-SunSoft response to the OMG Object Services RFI (OMG TC Document 92.2.10).

Basic, Flexible Services

The services are designed to do one thing well and are only as complicated as they need to be. Individual services are by themselves relatively simple yet they can, by virtue of their structuring as objects, be combined together in interesting and powerful ways.

For example, the event and life cycle services, plus a future relationship service, may play together to support graphs of objects. Object graphs commonly occur in the real world and must be supported in many applications. A functionally-rich Folder compound object, for example, may be constructed using the life cycle, naming, events, and future relationship services as “building blocks.”

Generic Services

Services are designed to be generic in that they do not depend on the type of the client object nor, in general, on the type of data passed in requests. For example, the event channel interfaces accept event data of any type. Clients of the service can dynamically determine the actual data type and handle it appropriately.

Allow Local and Remote Implementations

In general the services are structured as CORBA objects with OMG IDL interfaces that can be accessed locally or remotely and which can have local library or remote server styles of implementations. This allows considerable flexibility as regards the location of participating objects. So, for example, if the performance requirements of a particular application dictate it, objects can be implemented to work with a Library Object Adapter that enables their execution in the same process as the client.

Quality of Service is an Implementation Characteristic

Service interfaces are designed to allow a wide range of implementation approaches depending on the quality of service required in a particular environment. For example, in the Event Service, an event channel can be implemented to provide fast but unreliable delivery of events or slower but guaranteed delivery. However, the interfaces to the event channel are the same for all implementations and all clients. Because rules are not wired into a complex type hierarchy, developers can select particular implementations as building blocks and easily combine them with other components.

Objects Often Conspire in a Service

Services are typically decomposed into several distinct interfaces that provide different views for different kinds of clients of the service. For example, the Event Service is composed of *PushConsumer*, *PullSupplier* and *EventChannel* interfaces. This simplifies the way in which a particular client uses a service.

A particular service implementation can support the constituent interfaces as a single CORBA object or as a collection of distinct objects. This allows considerable implementation flexibility. A client of a service may use a different object reference to communicate with each distinct service function. Conceptually, these "internal" objects *conspire* to provide the complete service.

As an example, in the Event Service an event channel can provide both *PushConsumer* and *EventChannel* interfaces for use by different kinds of client. A particular client sends a request not to a single "event channel" object but to an object that implements either the *PushConsumer* and *EventChannel* interface. Hidden to all the clients, these objects interact to support the service.

The service designs also use distinct objects that implement specific service interfaces as the means to distinguish and coordinate different clients without relying on the existence of an object equality test or some special way of identifying clients. Using the event service again as an example, when an event consumer is connected with an event channel, a new object is created that supports the *PullSupplier* interface. An object reference to this object is returned to the event consumer which can then request events by invoking the appropriate operation on the new "supplier" object. Because each client uses a different object reference to interact with the event channel, the event channel can keep track of and manage multiple simultaneous clients. An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.

Use of Callback Interfaces

Services often employ callback interfaces. Callback interfaces are interfaces that a client object is required to support to enable a service to *call back* to it to invoke some operation. The callback may be, for example, to pass back data asynchronously to a client.

Callback interfaces have two major benefits:

- They clearly define how a client object participates in a service.
- They allow the use of the standard interface definition (OMG IDL) and operation invocation (object reference) mechanisms.

Assume No Global Identifier Spaces

Several services employ identifiers to label and distinguish various elements. The service designs do not assume or rely on any global identifier service or global id spaces in order to function. The scope of identifiers is always limited to some context. For example, in the naming service, the scope of names is the particular naming context object.

In the case where a service generates ids, clients can assume that an id is unique within its scope but should not make any other assumption.

Finding a Service is Orthogonal to Using It

Finding a service is at a higher level and orthogonal to using a service. These services do not dictate a particular approach. They do not, for example, mandate that all services must be found via the naming service. Because services are structured as objects there does not need to be a special way of finding objects associated with services - general purpose finding services can be used. Solutions are anticipated to be application and policy specific.

Interface Style Consistency

Use of Exceptions and Return Codes

Throughout the services, exceptions are used exclusively for handling exceptional conditions such as error returns. Normal return codes are passed back via output parameters. An example of this is the use of a DONE return code to indicate iteration completion.

Explicit Versus Implicit Operations

Operations are always explicit rather than implied (e.g., by a flag passed as a parameter value to some “umbrella” operation). In other words, there is always a distinct operation corresponding to each distinct function of a service.

Use of Interface Inheritance

Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by “normal” clients.

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- DSTC Pty Ltd (Cooperative Research Centre for Enterprise Distributed Systems Technology)
- Baltimore Technologies PLC

Overview 1

Contents

This chapter contains the following topics.

1.1 Introduction

A Public Key Infrastructure (PKI) is a collection of components or entities for the issuance, management, and revocation of digital certificates. Public key technology, although not new, does have promise as a basis for a flexible method of providing security in an online and distributed environment. For public key technology to reach this potential it must be possible to bind an identity to that of a public/private key pair (digital certificates) and then subsequently manage these using a PKI.

This document provides interfaces and operations in CORBA IDL to support the functionality of a PKI. It describes a generic interface that allows standards to be implemented behind these interfaces and operations. The specification also takes into consideration the possibility of specific CORBA extensions being designed at a later date that utilize specific technology within the CORBA framework. The interfaces provided in this document describe a standard method of interacting with PKI entities in a CORBA environment.

1.2 PKI Definitions

The following sections describe major components of a PKI. These are not specific to this specification nor to any specific existing standard, but are common PKI terms and components.

1.2.1 PKI User

A PKI user refers to human users as well as applications and hosts that may also use a PKI functionality.

1.2.2 Certificate

A certificate is a structured electronic document that binds some information to a public/private key pair and is digitally signed by a trusted third party called a *Certification Authority* or commonly referred to as a CA. This document will use the term CA throughout.

1.2.3 Certificate Revocation List (CRL)

A Certificate Revocation List (CRL) is a structured electronic document signed by a CA that lists any certificates previously issued by that CA that are now revoked. A certificate may be revoked for reasons that may include, but is not limited to, an entity or person changing or leaving a particular role or a private key being compromised. A CRL is issued on a periodic basis with the period determined by policy of the CA. A revoked certificate will remain on a CRL until the validity period of the certificate expires.

1.2.4 Certificate and CRL Repository

A repository is a service provided for the storage and retrieval of certificates and CRLs.

1.2.5 Certification Authority (CA)

A Certification Authority (CA) performs a number of functions relating to the issuance and management of public key certificates. These include:

- Accepting and verifying requests for certificates
- Revoking certificates and issuing CRLs
- Servicing requests for certificate status information
- Key management issues such as re-keying and re-certification

The CA may also certify the keys of other CAs so that the PKI can scale to multiple domains.

1.2.6 Registration Authority (RA)

A Registration Authority (RA) accepts requests for certificates on behalf of a CA and verifies the binding between the public/private key pair and the attributes being certified. Typically one or more RAs exist to provide a means for scaling a PKI within a single management domain. The relationship between the RAs and the CA is similar to the relationship between bank branches and the bank. While the branches are the “face” of the organization, the bank has the ultimate authority for the granting of transactions. So a request for a certificate may be made on a particular RA, the RA may verify Proof Of Possession (POP) of the private key and then request the certificate from the CA. The certificate obtained is from the CA but the RA provides the point of contact and may perform functions such as POP and checking authentication based on policy.

1.2.7 Online Certificate Status Service

This is a service used to determine the status of a certificate without the use of CRLs. Since CRLs can only be issued periodically, any revocation during this period is not known until the next issue of the CRL. Essentially this provides an online service to check the validity of a particular certificate and hence a more timely method of obtaining status information.

1.3 Specification Overview

This specification describes interfaces, constants, and constructs for interacting with a PKI through CORBA objects. The general design fits around existing standards and implementations and defines generic interfaces for the interaction with PKI components. This allows (but is not limited to) the wrapping of existing implementations. The major interfaces of this specification are shown (in rectangular boxes) in Figure 1-1 on page 1-4.

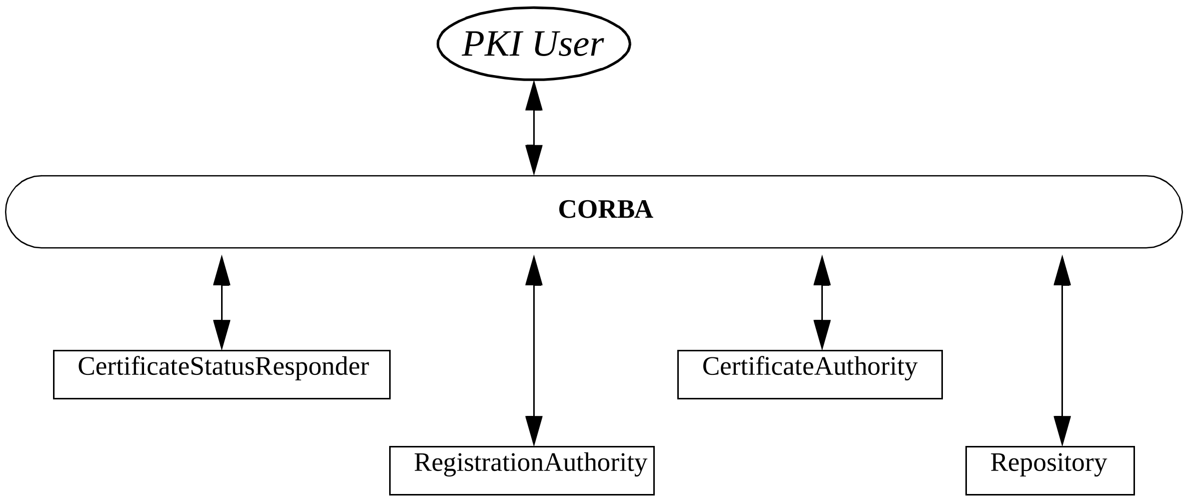


Figure 1-1 PKI Overview

There are other interfaces defined that are used to assist in the interaction with the above interfaces and allow for asynchronous messaging. These interfaces are described later in more detail. The functions of the major interfaces relate to the corresponding descriptions in Section 1.2, "PKI Definitions," on page 1-2.

1.3.1 PKI Module

This module describes common type definitions and constants that are used by the other modules.

1.3.2 PKIAuthority

This module outlines 14 interfaces for interacting with PKI authorities through the management of certificates. The major interfaces are *CertificateAuthority*, *RegistrationAuthority*, and *CertificateStatusResponder*. The rest are employed to maintain the interactive and asynchronous behavior that is typical using a PKI.

1.3.3 PKIRepository

This module provides interfaces and operations to store and retrieve certificates and CRLs.

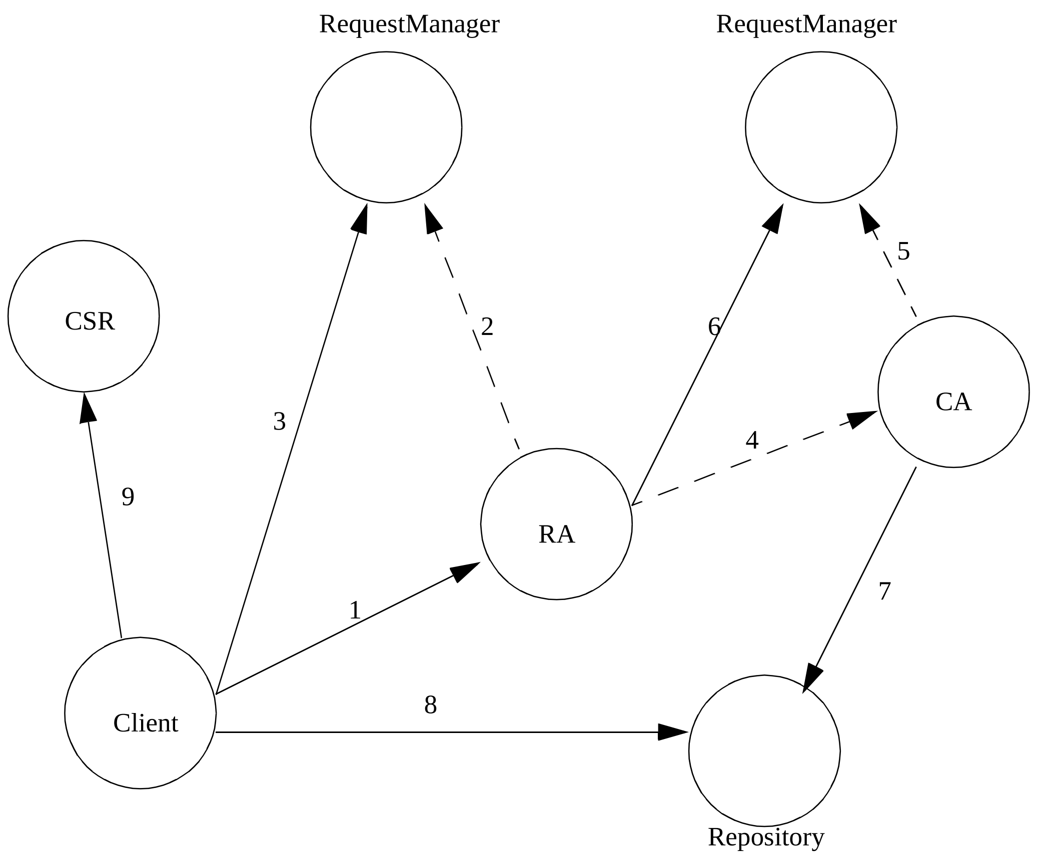
1.4 General PKI Usage Overview

This section describes some typical usage scenarios for interaction with the interfaces described in this specification. As mentioned earlier there is the potential for communication between PKI users to have an interactive and asynchronous nature. The asynchronicity comes from the fact that an authority will have policy regarding the issuance and management of certificates that may involve some out of band process (e.g., a phone call or email message). Complicating this even further is that the CA may need to interact with the clients to obtain more information and so a single invocation may not be sufficient to complete a particular request. For example a certificate request to an authority may require that POP is performed using a challenge response mechanism. This requires that the client decrypts a challenge and returns the result to the authority.

In addressing the interactive nature of PKI messaging additional interfaces have been added. These are the **RequestManager** interfaces. A **RequestManager** object is created by the authority. This is then used for any subsequent operations and status enquiries for a particular request. Asynchronous behavior can be addressed by using Asynchronous Method Invocation (AMI) described in CORBA/IIOP 2.4.2 document 2001-02-33, Chapter 22 CORBA Messaging.

1.4.1 Overall View

Typical operations that are expected from the set of specified interfaces are shown in Figure 1-2 on page 1-6. This diagram is included to show an overall structure of the specified interfaces.



1) Client makes a request to an RA, (similar if a client requests directly to a CA).
2) RA creates a RequestManager object.
3) Client using RequestManager to get results.
4,5,6) RA performs client operations to make request on CA.
7) CA access to Repository.
8) Client access to Repository
9) Client access to CertificateStatusResponder.

Figure 1-2 Interface Structure

1.4.2 Provider Information

Before a client makes requests to an authority it can obtain general details about what the authority provides. This may include general details such as version and vendor as well as specifics about supported types including whether an authority can handle callbacks in addition to polling.

1.4.3 Certificate Request

The following example is that of a client making a request to have a certificate issued by a CA or RA.

- After constructing a certificate request message, using a supported standard, the message can be encapsulated and invoked through the **request_certificate** operation. This returns a reference to a **RequestCertificateManager** object.
- The client can then call either **status** or **get_certificate_request_result** operations. If using the status operation, then the results will have to be obtained through an extra get result operation. Results can then be processed.
- After processing, if more interaction is required (for example, proving possession of the private key through a challenge response), then this is made through the manager object using the **continue_request_certificate** operation until a success (or failure) is reached.

1.5 General Repository Usage Overview

The PKI repository is defined as a service provided for the storage and retrieval of certificates, CRL's, and certificate pairs (collectively termed PKI values herein). Such PKI values are bound within the repository to a PKI principal, or user of PKI services. A PKI principal has some form of identifying name that distinguishes that principal within the repository. For example, there may be multiple certificates bound to the principal "Bob" within the repository. The PKI repository is designed to be conformant with respect to existing standards (specifically X.500 and LDAP), yet flexible enough to allow implementation using other services (e.g., databases, flat files).

An entry for a principal in the repository is assumed to have a number of attributes attached to it, where such attributes contain one or more values. Attributes are given a name, which facilitates the efficient search of the repository for specific values of a principal matching a particular attribute. For example, the CRL for the principal "BobCA" may be stored under an attribute with a name of "crl;binary". Thus, the attribute with the name "crl;binary" is used when finding CRLs for "BobCA."

In most cases, the repository implementor will have default attribute names for storing and retrieving PKI values, and clients should not have to specify exactly which attribute names are to be used when storing and retrieving PKI values in the repository. In the above example, the repository implementator may specify that the default attribute to use for storing CRLs for a principal is the attribute with the name "crl;binary". In such cases, the client only needs to provide the principal and the CRL to the repository. It is assumed that this will suffice for most clients and repository implementations (in particular, those implementations that use LDAP). Provision has been made for repository operations that allow the client to specify the particular attribute under which a given PKI value may be bound to a principal, and for determining the default attribute the repository implementor will use in specific cases.

1.6 Design Rationale

The design describes interfaces that have generic functionality that can support different underlying PKI standards. The wrapping of existing standards is an important issue with regard to this submission. There is also a reliance on other CORBA services for some functionality, primarily for security. Our goal was to consider functionality requirements to meet those of the RFP but also to meet those of *Internet X.509 PKI Certificate Management Protocols* IETF RFC 2510 and *Internet X.509 PKI Online Certificate Status* Protocol IETF RFC 2560.

1.6.1 Encoding to Representation Granularity

In this design a significant decision was made as to how best to represent certain data structures in CORBA. This is significant in this case because there are standards already defined and implemented that must be considered. Integrating to handle these standards is a significant issue being addressed by this specification. As a result, existing standards have different methods of encoding these structures so that they can be transported between entities. Encoding of these in the general case is encoding rules of ASN.1. This means that these types of structures can be represented as **PKI::Opaque** (i.e., a sequence of bytes).

The specification also allows for the possible future use of CORBA **valuetype**s. The use of **valuetype**s for representing types including a specific CORBA certificate would be useful. This specification allows for this by using an **any** type for the actual representation. The following IDL snippets demonstrate the design for encapsulating an outside encoded representation as well as a specific CORBA representation in a typesafe manner.

```
struct RepresentationType {
   EncodingType encoding_type;
   Opaque data;
};

struct Certificate {
   CertificateType certificate_type;
   any representation_type;
};
```

This specification defines and recommends the use of the **RepresentationType** for cases where it is logical that the representation is an encoded sequence of bytes. The **RepresentationType** allows for the tagging of the specific encoding type. The above sample IDL shows the **PKI::Certificate** type where the actual representation is implied to be that of the **RepresentationType** type, but it could be a valuetype for a case where a specific CORBA representation was defined. Similar situations to this example occur throughout the PKI module of the specification.

1.6.2 Asynchronous and Interactive Messaging

A significant design decision for this specification was in addressing the potential for asynchronous behavior combined with the potential for a level of interactivity between client and target entities. Each certification domain will have its own policy with which to manage certificate functionality. Depending on this policy it is possible for significant delays to occur between an initial request and a returned result due to the possible need for an out of band exchange. For example, a CA may require that a phone call or some interaction via email is made as part of the authentication process adding a significant delay. A synchronous invocation may block during this delay. This can be handled using an AMI aware ORB or simply ignored and wait for the invocation to be returned.

The potential for interactivity between an authority (CA or RA) and a client is also possible. An example of this interactivity might be where a certificate request has been made using a public key, the authority requires assurance that the client is in possession of the associated private key and policy dictates the use of a challenge response. This will require an extra exchange of messages and that the client may also be directly involved (by needing to supply a passphrase to unlock the private key). This interactivity for a request is addressed using the **RequestManager** interfaces. When a request is initiated a **RequestManager** object reference is returned, and this is used to perform further interaction, status checking, or to return results for that particular request.

The interaction of client and authority entities in a PKI domain is typically a combination of both an interactive dialogue, with state being maintained on the server side, combined with asynchronous messaging behavior. The **RequestManager** interface is created by the authority and encapsulates everything that relates to a particular request. The client entity receives a reference to this interface after an initial request and continues to use it for as long as the request is outstanding.

Resources related to **RequestManager**s can be reclaimed by the CertificateAuthority or RegistrationAuthority after either a status of either **PKISuccess**, **PKISuccessWithWarning** or **PKIFailed**. For a status of **PKISuccessAfterConfirm** the resources can be reclaimed after the **confirm_content()** operation has been invoked.

1.6.3 Repository

The repository is a service where information can be stored and retrieved. Primarily this is used for storage and retrieval of principal information such as certificates. It is also commonly used to store information such as Certificate Revocation Lists (CRLs). Effectively there are 2 definitions of repository interfaces in this specification. There is a simple interface described in PKIRepository and also a more intricate interface that is designed to be able to interact with repositories that are based around X500 and LDAP implementations. The simplified version is hoped to be the more commonly used interface allowing implementors and clients to interact with ease. However since the backend could well be LDAP or X500 in existing services that may be wrapped by these interfaces the PKIExtension module allows for this interaction. The operations in PKIRepository are simple and self explanatory but the PKIExtension module contains a more detailed approach.

The PKIExtension module describes a data storage service which generally has a schema that mandates the form and the content of the data stored therein. As much as possible, the type of repository implementation, and the exact details of the schema that oversees the data storage service, should be hidden from the client of the PKI repository service. In general, when a client wishes to publish information in the repository, it is assumed that the repository implementation has enough information to create the appropriate entry in the underlying data storage service according to the back-end schema. However, it may be the case that a repository implementation cannot gather the required information in order to create an entry for a principal when a request is made by the client to store information in the repository. For example, a database implementation of the PKI repository may require that all entries contain a value for the “favorite milkshake” field. In such cases, the repository implementation may ask for further information from the client. The **PKIPrincipal** type in the IDL allows the client to pass additional attribute information as required.

```
struct PKIPrincipal {
    PKIName name;
    PKIAttributeList attributes;
};
```

In most cases, the client will pass a **PKIPrincipal** construct to the repository with no attribute information. This is based on the assumption that there is already an entry for the given principal in the repository, or that the repository can create such an entry if this is the case. Clients should only pass attribute information within the **Principal** type if the repository has requested such information due to schema problems.

The PKI repository design allows the client to obtain the schema of the repository in order to present any additional attribute information required by the repository implementation. The **Schema** type is used to provide the client with two classes of information: information on attributes (OID, name, description, syntax, etc.) and information on syntaxes (OID, description). Given such a schema, a client may deduce the necessary values for attributes that are missing or incorrectly supplied. For example, if the repository notifies the client that a value for the “favorite milkshake” attribute is required, then the client may inspect the schema to lookup the attribute definition for “favorite milkshake,” find the syntax definition to see how a “favorite milkshake” value should be presented, and present that attribute information back to the repository within the **PKIPrincipal** structure. Each information class is represented within the schema as a collection of attributes (name-to-value bindings). A name is defined to be a string, while a value can be any type (including another collection of attributes).

The attribute list provided within the schema for attribute definitions is assumed to contain the name of each attribute used by the repository back-end. The value attached to each name is itself an attribute list, with names as defined in IETF RFC2252 for AttributeTypeDescriptions (“OID,” “NAME,” “DESCR,” “SYNTAX,” etc.). The value attached to each name is a string whose value is interpreted as defined by IETF RFC2252 (for example, the string attached to “OID” would represent an object identifier value such as “1.2.3”). The choice of which AttributeTypeDescription names to provide within the attribute list is up to the repository implementor, although they *should* provide at least the names “NAME,” “DESCR,” and “SYNTAX.”

The attribute list provided within the schema for syntax definitions is assumed to contain names that represent the object identifier of each syntax used within the attribute type definitions. The value attached to each name is itself an attribute list, with names defined in IETF RFC2252 for SyntaxDescriptions (“OID” and “DESCR”). The value attached to each of these names is a string whose value is interpreted as defined by IETF RFC2252. The choice of which SyntaxDescription names to provide within the attribute list is up to the repository implementor, although they SHOULD provide at least the name “DESCR.”

1.6.4 Provider Details

There are operations added to the interfaces that provide details about a particular implementation. This design decision was based around the fact that different underlying implementations may support different type formats and encodings. For example a particular CA may only support ASN.1 DER encoded X.509 certificates and so a client entity will need to query the CA and determine this detail. This is pertinent in the case of a PKI, as a CA is often authoritative in a particular domain and so a client may not have the choice to be able to choose its own CA based solely on supported types but be directed to use a particular one.

1.7 Proof of Concept

At the time of submission this design is currently being prototyped. The current status of this prototype demonstrates that the IDL is usable and can be implemented. The IDL is known to be parsed by at least one IDL compiler.

PKI Interfaces 2

Contents

This chapter contains the following topics.

2.1 Introduction

This chapter describes the basic interfaces and some important constructs and type definitions that are relevant to the specification.

2.2 Module PKI

This module declares type definitions used by both the **PKIAuthority** and **PKIRepository** modules. This section describes some of the particularly important constructs for clarity in understanding the interface operations in the rest of this chapter. The complete IDL is included in Appendix A.

2.2.1 PKIStatus Constants

Status constants are returned indicating the current status of a request.

typedef unsigned long PKIStatus;

2.2.1.1 *PKISuccess*

const PKIStatus PKISuccess = 0;

PKISuccess indicates that the current transaction is now complete without any more invocations required.

2.2.1.2 *PKISuccessWithWarning*

const PKIStatus PKISuccessWithWarning = 1;

PKISuccessWithWarning indicates that the client has received something similar to what was asked for. It is up to the client to ascertain the differences. This may for example be a certificate that varies in some way from the request such as the validity period may be different to that requested.

2.2.1.3 *PKIContinueNeeded*

const PKIStatus PKIContinueNeeded = 2;

PKIContinueNeeded indicates that the current part of the transaction is complete but the actual end result has not yet been reached. This means that another invocation is required most likely requiring some additional information.

2.2.1.4 *PKIFailed*

const PKIStatus PKIFailed = 3;

PKIFailed indicates that a failure has occurred and the transaction should be terminated.

2.2.1.5 *PKIPending*

const PKIStatus PKIPending = 4;

PKIPending indicates that the transaction is in a transitional period pending some result. This state occurs during the period before either a transaction is complete or a continue is required.

2.2.1.6 *PKISuccessAfterConfirm*

const PKIStatus PKISuccessAfterConfirm = 5;

PKISuccessAfterConfirm indicates that the transaction is complete but the PKIAuthority requires that a confirmation message is sent using **RequestManager.confirm_content()** operation. For example this might occur in the case where the CA may revoke the issued certificate if a confirm is not made as the CA may presume that the client could not decrypt the message as a way of providing proof of possession (POP) of the private key.

2.2.2 EncodingType

```
typedef unsigned long EncodingType;
   const EncodingType UnknownEncoding = 0;
   const EncodingType DEREncoding = 1;
   const EncodingType BEREncoding = 2;
   const EncodingType Base64Encoding = 3;
   const EncodingType SExprEncoding = 4;
   const EncodingType CustomEncoding = 0x8000;
```

The EncodingType is a type used to describe the method of encoding used to encode the original PKI structure to an Opaque type. The general case will be ASN.1 DER (Distinguished Encoding Rules).

2.2.3 Opaque

```
   typedef sequence <octet> Opaque;
```

The **Opaque** type is used to represent encoded structures as a sequence of bytes.

2.2.4 EncodedData

```
   struct EncodedData {
      EncodingType encoding_type;
      Opaque data;
   };
```

This construct is defined to be able to represent encoded structures in a type safe manner. This is recommended for implementations that are currently defined and represent structures using ASN.1 encoding rules.

2.2.5 CertificateType

```
typedef unsigned long CertificateType;
   const CertificateType UnknownCertificate = 0;
   const CertificateType X509v1Certificate = 1;
   const CertificateType X509v2Certificate = 2;
   const CertificateType X509v3Certificate = 3;
   const CertificateType PGPCertificate = 4;
   const CertificateType SPKICertificate = 5;
   const CertificateType X509v1AttributeCertificate = 6;
   const CertificateType CustomCertificate = 0x8000;
```

The CertificateType is used to explicitly describe the type of certificate that has been encoded. Some examples of certificate types are the X509 versions of certificate (version 3 being the most common in use), Pretty Good Privacy (PGP) certificates or Simple Public Key Infrastructure (SPKI) certificates.

2.2.6 EncodingType

```
typedef unsigned long EncodingType;
   const EncodingType UnknownEncoding = 0;
   const EncodingType DEREncoding = 1;
   const EncodingType BEREncoding = 2;
   const EncodingType Base64Encoding = 3;
   const EncodingType SExprEncoding = 4;
   const EncodingType CustomEncoding = 0x8000;
```

The EncodingType describes the way in which the byte representation is encoded. This is used to explicitly name the encoding method used. Some examples are ASN.1 Distinguished Encoding Rules (DER), ASN.1 Basic Encoding Rules (BER) ar perhaps Base 64 encoding.

2.2.7 AuthorityInfoType

```
typedef unsigned long AuthorityInfoType;
   const AuthorityInfoType UnknownMessage = 0;
   const AuthorityInfoType PKIXCMPGeneralMessage = 1;
   const AuthorityInfoType CustomMessage = 0x8000;
```

The AuthorityInfoType is used to describe the type of a message that is being sent/received by an authority. An example type for this is a PKIX Certificate Management Protocol (CMP) general message format.

2.2.8 Certificate

```
valuetype Certificate {
      private CertificateType certificate_type;
      private EncodedData data;
   };
```

This is the construct defined to represent a certificate in CORBA.

Fields

certificate_type	Describes the certificate type used, such as X509V1, X509V2, X509V3, PGP, SPKI.
data	This field contains a representation of the Certificate held in an EncodedData structure.

2.2.9 CRL

This construct is the representation of a Certificate Revocation List.

```
valuetype  CRL {
        private CRLType crl_type;
        private EncodedData data;
    };
```

Fields

crl_type	The type of CRL such as X509V1CRL, X509V2CRL, X509V1ARL.
data	This field contains a representation of the CRL held in an EncodedData structure.

2.2.10 CertificateRequest

The construct used to represent an encoded certificate request message.

```
valuetype CertificateRequest {
        private CertificateRequestType cert_request_type;
        private EncodedData data;
    };
```

Fields

cert_request_type	The type of certificate request message such as PKCS10, PKIXCRMF, PKIXCMC.
data	This field contains a representation of the CertificateRequest held in an EncodedData structure.

2.2.11 CertificateStatusRequest

```
valuetype CertificateStatusRequest {
        private CertificateStatusRequestType type;
        private EncodedData data;
    };
```

Fields

type	The type of certificate status request such as OCSP.
data	This field contains a representation of the CertificateStatusRequest held in an EncodedData structure.

2.2.12 *CertificateStatusResponse*

```
valuetype CertificateStatusResponse {
    private CertificateStatusResponseType type;
    private EncodedData data;
  };
```

Fields

type	The type of certificate status response such as OCSP.
data	This field contains a representation of the CertificateStatusResponse held in an EncodedData structure.

2.2.13 *Exceptions*

2.2.13.1 *UnsupportedTypeException*

```
exception UnsupportedTypeException {
  string description;
};
```

Exception reporting either the Certificate, CertificateRequest, CertificateStatusRequest, etc. supplied is not a supported type by the PKIAuthority interface.

2.2.13.2 *UnsupportedEncodingException*

```
exception UnsupportedEncodingException {
  string description;
};
```

Exception reporting either the Certificate, CertificateRequest, CertificateStatusRequest, etc. supplied is using an unsupported encoding type.

2.2.13.3 *MalformedDataException*

```
exception MalformedDataException {
  string description;
};
```

Exception reporting either the Certificate, CertificateRequest, CertificateStatusRequest, etc. supplied is in some way malformed and cannot be interpreted.

2.2.13.4 *UnexpectedContinueException*

```
exception UnexpectedContinueException {
    string description;
};
```

Exception reporting either the Certificate, CertificateRequest, CertificateStatusRequest, etc. supplied is attempting an unnecessary continue operation.

2.3 *Module PKIAuthority*

2.3.1 *Interface RegistrationAuthority*

2.3.1.1 *get_provider_info*

Used to obtain a standard set of types supported by this authority.

```
AuthorityProviderInfo get_provider_info();
```

Return Value

AuthorityProviderInfo structure holding descriptions of supported types.

2.3.1.2 *get_authority_info*

Used for passing general messages between client entity and authority. For example this may provide a method for a client to determine the authentication policy of the authority.

```
PKI::PKIStatus get_authority_info(
         in PKI::AuthorityInfo  in_authority_info,
         out PKI::AuthorityInfo out_authority_info
         )
         raises(UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
```

Parameters

in_authority_info	The encoded message input to authority.
out_authority_info	The encoded returned message from authority.

Return Value

Status value

2.3.1.3 *request_certificate*

Called to make a request for a certificate from an authority such as a Certificate Authority (CA) or Registration Authority (RA).

```
RequestCertificateManager request_certificate
        (in PKI::CertificateRequest certificate_request)
        raises(UnsupportedTypeException,UnsupportedEncodingException,
             MalformedDataException);
```

Parameters

certificate_request	**PKI::CertificateRequest** structure containing details of the clients request.

Return Value

RequestCertificateManager object reference to extract details regarding the particular request, continue interaction, and obtain results.

2.3.1.4 *request_revocation*

Called to request revocation of a certificate from a (CA) or (RA).

```
RequestRevocationManager request_revocation
        (in PKI::CertRevRequest     cert_rev_request)
        raises(UnsupportedTypeException,UnsupportedEncodingException,
             MalformedDataException);
```

Parameters

cert_rev_request	**PKI::CertRevRequest** structure containing details of the client’s request for certificate revocation.

Return Value

RequestRevocationManager object reference used to extract details pertaining to the request continue interaction, and obtain results.

2.3.1.5 *request_key_update*

Called to request key update of a certificate from a (CA) or (RA).

```
RequestKeyUpdateManager request_key_update
        (in PKI::CertificateRequest   key_request)
        raises(UnsupportedTypeException,UnsupportedEncodingException,
             MalformedDataException);
```

Parameters

key_request	**PKI::CertificateRequest** structure containing details of the client's request for key update.

Return Value

RequestKeyUpdateManager object reference used to extract details pertaining to the request, continue interaction, and obtain results.

2.3.1.6 *request_key_recovery*

```
RequestKeyRecoveryManager request_key_recovery
        (in PKI::CertificateRequest   key_request)
        raises(UnsupportedTypeException,UnsupportedEncodingException,
             MalformedDataException);
```

Parameters

key_request	**PKI::CertificateRequest** structure containing details of the client's request for key recovery.

Return Value

RequestKeyRecoveryManager object reference that can be used to extract details pertaining to the request, continue interaction, and obtain results.

2.3.2 *Interface CertificateAuthority*

Interface defining operations that can be performed on a **CertificateAuthority** object. There is IDL inheritance of the **RegistrationAuthority** interface.

2.3.2.1 *get_ca_certificate*

Returns the certificates of the **CertificateAuthority**.

```
PKI::PKIStatus get_ca_certificate(
        out PKI::CertificateList certificate_list)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
```

Parameters

certificate_list	List of certificates for CA.

Return Value

PKI::PKIStatus indicating the status of the request.

2.3.2.2 get_CRL

Return the current CRL of the **CertificateAuthority**.

PKI::PKIStatus get_crl (out PKI::CRL crl);

Parameters

crl	The CRL published by the **CertificateAuthority**.

Return Value

PKI::PKIStatus indicating the status of the request.

2.3.2.3 get_certificate_status_responder

Return a reference to certificate status responder.

CertificateStatusResponder get_certificate_status_responder();

Return Value

Reference to **CertificateStatusResponder** object for the CA.

2.3.2.4 get_repository

Return a reference to the repository that the CA uses to store certificates, CRLs, etc.

```
PKIRepository::Repository get_repository()
        raises(PKIRepository::RepositoryError);
```

Return Value

PKIRepository::Repository object reference.

2.3.3 Interface RequestManager

Generic base interface for a manager object. A manager object is a target side object that is used by the client to extract details, continue interaction, and to obtain results for a particular request.

2.3.3.1 status

A read only attribute representing the status of the transaction associated with this poller object.

readonly attribute PKI::PKIStatus status;

2.3.3.2 *transaction_ID*

readonly attribute long transaction_ID;

A read only attribute representing an identifier for a particular transaction. This attribute relates directly to existing PKI entities. Currently a transaction will be given some unique identifier that relates to a particular transaction with an authority. In the case of using CORBA the unique identifier is not directly required due to the use of a RequestManager object for each transaction. This attribute is supplied so that the identifier provided by the back end authority can be obtained by a CORBA client.

2.3.3.3 *confirm_content*

Operation to acknowledge negotiation is complete.

```
void confirm_content(in PKI::ConfirmData     confirm_data)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
```

Parameters

confirm_data	Message to confirm content is correct and received.

2.3.4 *Interface RequestCertificateManager*

Interface to extract details, continue interaction and extract results pertaining to a particular certificate request. Inherits operations and attributes from **RequestManager** interface.

2.3.4.1 *continue_request_certificate*

Used for continuing a certificate request that has already been initiated but requires more interaction to complete the request. An example of the use of this operation is for Proof Of Possession (POP) of the private key.

```
void continue_request_certificate
        (in PKI::RequestData                         request_data,
         in PKI::CertificateList                     certificates)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
```

Parameters

request_data	**PKI::RequestData** structure containing details for the continuation of the initial request.
certificates	List of certificates, possibly partially formed.

2.3.4.2 *get_certificate_request_result*

Obtains final or interim results of a particular request.

```
PKI::PKIStatus get_certificate_request_result
          (out  PKI::CertificateList    certificates,
           out PKI::ResponseData        response_data)
          raises (UnsupportedTypeException,UnsupportedEncodingException,
                 MalformedDataException);
```

Parameters

certificates	A list of certificates.
response_data	**PKI::ResponseData** structure containing details of the request thus far.

Return Value

PKI::PKIStatus indicating the status of the request.

2.3.5 *Interface RequestRevocationManager*

Interface to extract details, continue interaction and extract results pertaining to a particular revocation request. Inherits operations and attributes from **RequestManager** interface.

2.3.5.1 *continue_request_revocation*

Used for continuing a revocation request that has already been initiated but requires more interaction to complete the request.

```
 void continue_request_revocation
          (in PKI::RequestData     request_data)
          raises (UnsupportedTypeException,UnsupportedEncodingException,
                 MalformedDataException);
```

Parameters

request_data	**PKI::RequestData** structure containing details for the continuation of the initial request.

2.3.5.2 *get_request_revocation_result*

Obtains final or interim results of a particular request.

```
PKI::PKIStatus get_request_revocation_result
        (out PKI::CertRevResponse    cert_rev_response,
         out PKI::ResponseData       response_data)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
```

Parameters

cert_rev_response	**PKI::CertRevResponse** structure containing details of the response of the the revocation request.
response_data	**PKI::ResponseData** structure containing details of the request thus far for continuing the request.

Return Value

PKI::PKIStatus indicating the status of the request.

2.3.6 *Interface RequestKeyUpdateManager*

Interface to extract details, continue interaction, and extract results pertaining to a particular key update request. Inherits attributes and operations from **RequestManager** interface.

2.3.6.1 *continue_key_update*

Used for continuing a key recovery request that has already been initiated but requires more interaction to complete the request.

```
void continue_key_update
        (in PKI::RequestData                request_data,
         in PKI::Certificate                certificate)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
```

Parameters

request_data	**PKI::RequestData** structure containing details for the continuation of the initial request.
certificate	**PKI::Certificate**

2.3.6.2 *get_request_key_update_result*

Obtains final or interim results of a particular request.

```
PKI::PKIStatus get_request_key_update_result
        (out PKI::Certificate      certificate,
         out PKI::ResponseData        response_data)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
```

Parameters

certificate	The new certificate after key update.
response_data	**PKI::ResponseData** structure containing details of the request thus far.

Return Value

PKI::PKIStatus indicating the status of the request.

2.3.7 *Interface RequestKeyRecoveryManager*

Interface to extract details, continue interaction and extract results pertaining to a particular key recovery request. Inherits attributes and operations from **RequestManager** interface.

2.3.7.1 *continue_key_recovery*

Used for continuing a key recovery request that has already been initiated but requires more interaction to complete the request.

```
void continue_key_recovery
        (in PKI::RequestData     request_data)
         raises
(UnsupportedTypeException,UnsupportedEncodingException,
               MalformedDataException);
```

Parameters

request_data	**PKI::RequestData** structure containing details for the continuation of the initial request.

2.3.7.2 *get_request_key_recovery_result*

Obtains final or interim results of a particular request.

```
PKI::PKIStatus get_request_key_recovery_result
        (out PKI::ResponseData      response_data)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
```

Parameters

response_data	**PKI::ResponseData** structure containing details of the request thus far.

Return Value

PKI::PKIStatus indicating the status of the request.

2.3.8 *Interface CertificateStatusResponder*

Interface for an online certificate status responder.

2.3.8.1 *request_certificate_status*

Obtains details for the request of a certificate status from an online certificate status server.

```
PKI::PKIStatus request_certificate_status(
        in PKI::CertificateStatusRequest request,
        out PKI::CertificateStatusResponse response)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
```

Parameters

request	**PKI::CertificateStatusRequest** structure containing details of the request.
response	**PKI::CertificateStatusResponse** structure containing details of the return response.

Return Value

PKI::PKIStatus indicating the status of the request.

2.4 Module PKIRepository

2.4.1 PKIPrincipalValue

Valuetype supporting fields containing relevant information related to a particular principal.

```
valuetype PKIPrincipalValue {
    private string name;
    private PKI::CertificateList certificates;
    private PKI::CertificatePairList;
    private PKI::CRL crl;
    private PKI::CRL delta;
    private PKI::CRL arl;
  };
```

2.4.2 Interface Repository

Interface for storage and retrieval of certificates and CRLs.

2.4.2.1 RepositoryProviderInfo info

Attribute to return a valuetype containing information pertaining to this particular repository implementation.

2.4.2.2 publish

Enter a new PKIPrincipalValue into the repository.

```
void publish( in PKIPrincipalValue principal )
     raises ( DuplicatePrincipal, RepositoryError );
```

Parameters

principal	PKIPrincipalValue to be entered.

2.4.2.3 *locate*

Get a PKIPrincipalValue for a particular name.

```
PKIPrincipalValue locate ( in string name )
      raises ( UnknownPrincipal, RepositoryError );
```

Parameters

name	String name of desired principal to be located.

Return Value

PKIPrincipalValue of the specified name.

2.4.2.4 *delete*

Deletes a principal in the repository using name as the lookup key.

```
void delete ( in string name )
      raises ( UnknownPrincipal, RepositoryError );
```

Parameters

name	String name of desired principal to be deleted

2.4.2.5 *update*

Replaces an existing principal in the repository with the supplied PKIPrincipalValue object.

```
void update ( in PKIPrincipalValue principal )
      raises ( UnknownPrincipal, RepositoryError);
```

Parameters

principal	PKIPrincipalValue to be updated.

2.5 Module PKIExtension

2.5.1 Interface LDAPRepository

Interface for a repository for the storage and retrieval of certificates and CRLs.

2.5.1.1 get_provider_info

Get the provider info for this PKI repository.

```
RepositoryProviderInfo get_provider_info();
```

Return Value

RepositoryInfo construct containing general details relating the provider implemented repository.

2.5.1.2 get_schema

Called to retrieve details of the schema used for the particular repository.

```
Schema get_schema();
```

Return Value

Schema construct containing lists of attribute and syntax definitions.

2.5.1.3 publish_certificate

Publish a certificate for the given principal, under the attribute specified by the given attribute name.

```
void publish_certificate(
        in PKIPrincipal principal,
        in PKI::Certificate certificate, in string attr_name)
        raises (PKIRepository::UnknownPrincipal,
        PrincipalAttributeError,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the certificate is to be bound.
certificate	The certificate to be published.
attr_name	The name of the attribute under which this certificate is to be stored in the repository entry of the principal.

2.5.1.4 get_certificate

Get the certificate(s) associated with a given principal, under the attribute specified by the given attribute name. If there are no certificates bound to the given principal (i.e., the given attribute does not exist, or that attribute exists but has no certificate values), then a list of length 0 is returned.

```
PKI::CertificateList get_certificate(
        in PKIPrincipal principal, in string attr_name)
        raises (PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principle	The principal whose certificates are to be returned.
attr_name	The name of the attribute containing the certificate(s) in the repository entry of the given principal.

Return Value

The (possibly empty) list of certificates bound to the entry in the repository for the given principal, where such certificates (if any) are stored as values of the given attribute.

2.5.1.5 delete_certificate

Deletes the given certificate stored against the given principal under the attribute specified by the given name. How the given certificate is matched against stored certificates is implementation-dependent.

```
void delete_certificate(
        in PKIPrincipal principal,
        in PKI::Certificate certificate, in string attr_name)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal whose certificate is to be deleted.
certificate	The certificate to be deleted.
attr_name	The name of the attribute containing the certificate in the repository entry of the given principal.

2.5.1.6 publish_crl

Publish the given certificate revocation list for the given principal under the attribute specified by the given name.

```
void publish_crl(in PKIPrincipal principal, in PKI::CRL crl,
                             in string attr_name)
          raises(PKIRepository::UnknownPrincipal,
          PrincipalAttributeError,
          PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the CRL is bound.
crl	The certificate revocation list to be published.
attr_name	The name of the attribute under which this crl is to be stored in the repository entry of the principal.

2.5.1.7 *get_crl*

Get the CRL associated with a given principal, under the attribute specified by the given name.

```
PKI::CRL get_crl(in PKIPrincipal principal, in string attr_name)
          raises(PKIRepository::UnknownPrincipal,
          PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the CRL is bound.
attr_name	The name of the attribute under which this crl is to be stored in the repository entry of the principal.

Return Value

PKI::CRL structure containing the CRL.

2.5.1.8 *delete_crl*

Deletes the given CRL stored against the given principal under the attribute specified by the given name. How the given CRL is matched against stored CRL is implementation-dependent.

```
void delete_crl(in PKIPrincipal principal,
                            in PKI::CRL crl,in string attr_name)
          raises(PKIRepository::UnknownPrincipal,
          PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the CRL is bound.
crl	The certificate revocation list to be deleted.
attr_name	The name of the attribute under which this crl is stored in the repository entry of the principal.

2.5.1.9 *publish_certificate_pair*

Publish the given certificate pair for the given principal under the attribute specified by the given name.

```
void publish_certificate_pair(
        in PKIPrincipal principal, in PKI::CertificatePair certPair,
        in string attr_name)
        raises(PKIRepository::UnknownPrincipal,
        PrincipalAttributeError,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the certificate pair is bound.
certPair	The certificate pair to be published.
attr_name	The name of the attribute under which this certificate pair is to be stored in the repository entry of the principal.

2.5.1.10 *get_certificate_pair*

Get the certificate pair(s) associated with a given principal, under the attribute specified by the given name. If there are no certificate pair(s) bound to the given principal (i.e., the given attribute does not exist, or that attribute exists but has no certificate pair values), then a list of length 0 is returned.

```
PKI::CertificatePairList get_certificate_pair(
        in PKIPrincipal principal, in string attr_name)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the certificate pair is bound.
attr_name	The name of the attribute under which this certificate pair is stored in the repository entry of the principal.

Return Value

PKI::CertificatePairList containing the requested certificate pairs.

2.5.1.11 *delete_certificate_pair*

Deletes the given certificate pair stored against the given principal under the attribute specified by the given name. How the given certificate is matched against stored certificate pairs is provider implementation-dependent.

```
void delete_certificate_pair(
        in PKIPrincipal principal,
        in PKI::CertificatePair certificate_pair,
        in string attr_name)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the certificate pair is bound.
certificate_pair	The certificate pair to be deleted.
attr_name	The name of the attribute under which this certificate pair is stored in the repository entry of the principal.

2.5.1.12 *publish_user_certificate*

Publish a given certificate for the given principal under the attribute specified by the repository implementator as the default attribute for storing certificates to be interpreted as user certificates.

```
void publish_user_certificate(in PKIPrincipal principal,
                        in PKI::Certificate certificate)
        raises(PKIRepository::UnknownPrincipal,
        PrincipalAttributeError,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the certificate is bound.
certificate	The user certificate to be published.

2.5.1.13 *get_user_certificate*

Get the certificate(s) for the given principal under the attribute specified by the repository implementator as the default attribute for storing certificates to be interpreted as user certificates.

```
PKI::CertificateList get_user_certificate(in PKIPrincipal principal)
        raises(UnknownPrincipal,RepositoryError);
```

Parameters

principal	The principal to which the certificate is bound.

Return Value

PKI::CertificateList containing the list of requested user certificates.

2.5.1.14 *delete_user_certificate*

Delete the given certificate bound to the given principal under the attribute specified by the repository implementator as the default attribute for storing certificates to be interpreted as user certificates.

```
void delete_user_certificate(in PKIPrincipal principal,
                             in PKI::Certificate certificate)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the certificate is bound.
certificate	The user certificate to be deleted.

2.5.1.15 *publish_ca_certificate*

Publish a given certificate for the given principal under the attribute specified by the repository implementator as the default attribute for storing certificates to be interpreted as CA certificates.

```
void publish_ca_certificate(
        in PKIPrincipal principal,
        in PKI::Certificate certificate)
        raises(PKIRepository::UnknownPrincipal,
        PrincipalAttributeError,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the CA certificate is bound.
certificate	The certificate to be published.

2.5.1.16 *get_ca_certificates*

Get the certificate(s) bound to the given principal under the attribute specified by the repository implementor as the default attribute for storing certificates to be interpreted as CA certificates.

```
PKI::CertificateList get_ca_certificates(in PKIPrincipal principal)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the CA certificates are bound.

Return Value

PKI::CertificateList containing requested CA certificates.

2.5.1.17 *delete_ca_certificate*

Delete the given certificate bound to the given principal under the attribute specified by the repository implementator as the default attribute for storing certificates to be interpreted as CA certificates.

```
void delete_ca_certificate(in PKIPrincipal principal,
                        in PKI::Certificate certificate)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);;
```

Parameters

principal	The principal to which the CA certificate is bound.
certificate	The certificate to be deleted.

2.5.1.18 *publish_default_crl*

Publish the given CRL for the given principal under the attribute specified by the repository implementor as the default attribute for storing CRLs.

```
void publish_default_crl(in PKIPrincipal principal, in PKI::CRL crl)
      raises(PKIRepository::UnknownPrincipal,
      PrincipalAttributeError,
      PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the CRL is bound.
crl	The certificate revocation list to be published.

2.5.1.19 get_default_crl

Get the CRL bound to the given principal under the attribute specified by the repository implementor as the default attribute for storing CRLs.

```
PKI::CRL get_default_crl(in PKIPrincipal principal)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the CRL is bound.

Return Value

The PKI::CRL containing the requested CRL.

2.5.1.20 delete_default_crl

Delete the specified CRL bound to the given principal under the attribute specified by the repository implementor as the default attribute for storing CRLs. How the given CRL is matched against stored CRLs is provider implementation-dependent.

```
void delete_default_crl(in PKIPrincipal principal, in PKI::CRL crl)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the CRL is bound.
crl	The certificate revocation list to be deleted.

2.5.1.21 publish_default_certificate_pair

Publish the given certificate pair for the given principal under the attribute specified by the repository implementor as the default attribute for storing certificate pairs.

```
void publish_default_certificate_pair(in PKIPrincipal principal,
                                in PKI::CertificatePair certificate_pair)
        raises(PKIRepository::UnknownPrincipal,
        PrincipalAttributeError,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the certificate pair is bound.
certificate_pair	The certificate pair to be published.

2.5.1.22 *get_default_certificate_pair*

Get the certificate pair(s) bound to the given principal under the attribute specified by the repository implementor as the default attribute for storing certificate pairs.

```
PKI::CertificatePairList get_default_certificate_pair
                        (in PKIPrincipal principal)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the certificate pairs are bound.

Return Value

PKI::CertificatePairList containing requested certificate pairs.

2.5.1.23 *delete_default_certificate_pair*

Delete the specified certificate pair bound to the given principal under the attribute specified by the repository implementor as the default attribute for storing certificate pairs. How the given certificate pair is matched against stored certificate pairs is provider implementation-dependent.

```
void delete_default_certificate_pair(in PKIPrincipal principal,
                                in PKI::CertificatePair certificate_pair)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the certificate pairs are bound.
certificate_pair	The certificate pair to be deleted.

2.5.1.24 *publish_delta_crl*

Publish the given delta CRL for the given principal under the attribute specified by the repository implementor as the default attribute for storing delta CRLs

void publish_delta_crl(in PKIPrincipal principal, in PKI::CRL delta_crl);

Parameters

principal	The principal to which the delta CRL is bound.
delta_crl	The delta CRL to be published.

2.5.1.25 *get_delta_crl*

Get the delta CRL bound to the given principal under the attribute specified by the repository implementor as the default attribute for storing delta CRLs.

```
PKI::CRL get_delta_crl(in PKIPrincipal principal)
        raises(PKIRepository::UnknownPrincipal, PrincipalAttributeError,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the delta CRL is bound.

Return Value

PKI::CRL containing the requested delta CRL.

2.5.1.26 *delete_delta_crl*

Delete the specified delta CRL bound to the given principal under the attribute specified by the repository implementor as the default attribute for storing delta CRLs. How the given CRL is matched against stored CRLs is provider implementation-dependent.

```
void delete_delta_crl(in PKIPrincipal principal, in PKI::CRL delta_crl)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);
```

Parameters

principal	The principal to which the delta CRL is bound.
delta_crl	The delta CRL to be deleted.

2.5.1.27 *publish_arl*

Publish the given ARL for the given principal under the attribute specified by the repository implementor as the default attribute for storing ARLs.

```
void publish_arl(in PKIPrincipal principal, in PKI::CRL arl)
        raises(PKIRepository::UnknownPrincipal,
        PrincipalAttributeError,
        PKIRepository::RepositoryError);;
```

Parameters

principal	The principal to which the ARL is bound.
arl	The ARL to be published.

2.5.1.28 *get_arl*

Get the ARL bound to the given principal under the attribute specified by the repository implementor as the default attribute for storing ARLs.

```
PKI::CRL get_arl(in PKIPrincipal principal)
        raises(PKIRepository::UnknownPrincipal,
        PKIRepository::RepositoryError);;
```

Parameters

principal	The principal to which the ARL is bound.

Return Value

PKI::CRL containing the requested ARL.

2.5.1.29 *delete_arl*

Delete the specified ARL bound to the given principal under the attribute specified by the repository implementor as the default attribute for storing ARLs. How the given ARL is matched against stored ARLs is provider implementation-dependent.

```
void delete_arl(in PKIPrincipal principal, in PKI::CRL arl)
        raises(PKIRepository::UnknownPrincipal,
        RPKIRepository::epositoryError);
```

Parameters

principal	The principal to which the ARL is bound.
arl	The ARL to be deleted.

OMG IDL A

A.1 PKI

```
#ifndef __PKI_IDL
#define __PKI_IDL

#pragma prefix “omg.org”

module PKI {

    typedef sequence <octet> Opaque;

    //Certificate Types
    typedef unsigned long CertificateType;
    const CertificateType UnknownCertificate = 0;
    const CertificateType X509v1Certificate = 1;
    const CertificateType X509v2Certificate = 2;
    const CertificateType X509v3Certificate = 3;
    const CertificateType PGPCertificate = 4;
    const CertificateType SPKICertificate = 5;
    const CertificateType X509v1AttributeCertificate = 6;
    const CertificateType CustomCertificate = 0x8000;

    typedef sequence <CertificateType> CertificateTypeList;

    //Certificate Encoding Types
    typedef unsigned long EncodingType;
    const EncodingType UnknownEncoding = 0;
    const EncodingType DEREncoding = 1;
    const EncodingType BEREncoding = 2;
    const EncodingType Base64Encoding = 3;
    const EncodingType SExprEncoding = 4;
    const EncodingType CustomEncoding = 0x8000;

    // A representation type to deal with current existing PKI implementations
    // and standards.
```

```
struct ~~RepresentationType~~ EncodedData {
        EncodingType encoding_type;
        Opaque data;
};

typedef unsigned long AuthorityInfoType;
const AuthorityInfoType UnkownMessage = 0;
const AuthorityInfoType PKIXCMPGeneralMessage = 1;
const AuthorityInfoType CustomMessage = 0x8000;

struct AuthorityInfo {
        AuthorityInfoType authority_info_type;
        RepresentationType representation_type;
};

//
// Certificate information - used in both the Certificate definition
// and the PKIAuthority::RegistrationAuthorityProviderInfo definition.
//
struct CertificateInfo {
CertificateType certificate_type;
EncodingType encoding_type;
};
typedef sequence<CertificateInfo> CertificateInfoList;

//Certificate
valuetype Certificate {
        private CertificateType certificate_type;
        private EncodedData data;
};

typedef sequence <Certificate> CertificateList;

//CRL Types
typedef unsigned long CRLType;
const CRLType UnknownCRL = 0;
const CRLType X509v1CRL = 1;
const CRLType X509v2CRL = 2;
const CRLType X509V1ARL = 3;
const CRLType CustomCRL = 0x8000;

typedef sequence <CRLType> CRLTypeList;

// Information about a CRL
struct CRLInfo {
CRLType crl_type;
EncodingType encoding_type;
};

typedef sequence<CRLInfo> CRLInfoList;

valuetype  CRL {
        private CRLType crl_type;
        private EncodedData data;
```

```
};


//Certificate Request Type
typedef unsigned long CertificateRequestType;
const CertificateRequestType UnknownCertificateRequest = 0;
const CertificateRequestType PKCS10CertificateRequest = 1;
const CertificateRequestType PKIXCRMFCertificateRequest = 2;
const CertificateRequestType PKIXCMCCertificateRequest = 3;
const CertificateRequestType CustomCertificateRequest = 0x8000;

typedef sequence <CertificateRequestType> CertificateRequestTypeList;

// Information about a certificate request
struct CertificateRequestInfo {
      CertificateRequestType cert_request_type;
      EncodingType encoding_type;
};
typedef sequence<CertificateRequestInfo> CertificateRequestInfoList;

//Certificate Request
~~struct CertificateRequest {~~

valuetype CertificateRequest {
       private CertificateRequestType cert_request_type;
       private EncodedData data;
};


struct CertificatePair {
      Certificate forward;
      Certificate reverse;
};
typedef sequence<CertificatePair> CertificatePairList;

//ContinueType
typedef unsigned long ContinueType;
const ContinueType UnknownContinue = 0;
const ContinueType PKIXCMPContinue = 1;
const ContinueType PKIXCMCContinue = 2;
const ContinueType PKIXCMPConfirm = 3;
const ContinueType PKIXCMCConfirm = 4;
const ContinueType CustomContinue = 0x8000;

//Continue Structure
valuetype Continue {
      private ContinueType continue_type;
      private EncodedData data;
};

//ContinueData
// Request indicates from client to target message exchange
typedef Continue RequestData;

//ContinueResponse
```

```
// Response indicates from target to client message exchange
typedef Continue ResponseData;

// ConfirmData
typedef Continue ConfirmData;

//Certificate Revocation Type
typedef unsigned long CertRevocationType;
const CertRevocationType UnknownCertRevocation = 0;
const CertRevocationType PKIXCMPCertRevocation = 1;
const CertRevocationType PKIXCMCCertRevocation = 2;
const CertRevocationType CustomCertRevocation = 0x8000;

// Information about Certificate revocation
struct CertificateRevocationInfo {
      CertRevocationType cert_rev_type;
      EncodingType encoding_type;
};
typedef sequence <CertificateRevocationInfo> CertificateRevocationInfoList;

//Certificate Revocation

valuetype CertRevocation {
      private CertRevocationType cert_rev_type;
      private EncodedData data;
};

//Certificate Revocation Respone
typedef CertRevocation CertRevResponse;
typedef CertRevocation CertRevRequest;

//Key Recovery Type
typedef unsigned long KeyRecoveryType;
const KeyRecoveryType UnkownKeyRecovery = 0;
const KeyRecoveryType PKIXCMPKeyRecovery = 1;
const KeyRecoveryType PKIXCMCKeyRecovery = 2;
const KeyRecoveryType CustomKeyRecovery = 0x8000;

// Information about key recovery
struct KeyRecoveryInfo {
      KeyRecoveryType key_rec_type;
      EncodingType encoding_type;
};
typedef sequence <KeyRecoveryInfo> KeyRecoveryInfoList;

//Key Recovery Response
valuetype KeyRecResponse {
       private KeyRecoveryType key_recovery;
       private EncodedData data;
};

//OCSP
//Certificate status request type
typedef unsigned long CertificateStatusRequestType;
const CertificateStatusRequestType
```

```
        UnknownCertificateStatusRequestType = 0;
    const CertificateStatusRequestType
        OCSPCertificateStatusRequest = 1;
    const CertificateStatusRequestType
        CustomCertificateStatusRequest = 0x8000;

    //Type for certificate status requests
    valuetype CertificateStatusRequest {
        private CertificateStatusRequestType type;
        private EncodedData data;
    };


    //Certificate status response type
    typedef unsigned long CertificateStatusResponseType;
    const CertificateStatusResponseType
        UnknownCertificateStatusResponseType = 0;
    const CertificateStatusResponseType
        OCSPCertificateStatusResponse = 1;
    const CertificateStatusResponseType
        CustomCertificateStatusResponse = 0x8000;

    //Type for certificate status responses
    valuetype CertificateStatusResponse {
        private CertificateStatusResponseType type;
        private EncodedData data;
    };

    typedef unsigned long PKIStatus;
    const PKIStatus PKISuccess = 0;
    const PKIStatus PKISuccessWithWarning = 1;
    const PKIStatus PKIContinueNeeded = 2;
    const PKIStatus PKIFailed = 3;
    const PKIStatus PKIPending = 4;
    const PKIStatus PKISuccessAfterConfirm = 5;
    };
#endif
```

A.2 PKIAuthority

```
#ifndef __PKIAUTHORITY_IDL
#define __PKIAUTHORITY_IDL

#include <PKI.idl>
#include <PKIRepository.idl>

#pragma prefix “omg.org”


module PKIAuthority {

    // Forward declaration...
    interface CertificateStatusResponder;
```

```
        interface RequestManager;
        interface RequestCertificateManager;
        interface RequestRevocationManager;
        interface RequestKeyUpdateManager;
        interface RequestKeyRecoveryManager;

valuetype AuthorityProviderInfo {
        public string standardVersion;
        public string standardDescription;
        public string productVersion;
        public string productDescription;
        public string productVendor;
        public PKI::CertificateInfoList supportedCertificates;
        public PKI::CRLInfoList supportedCRLs;
        public PKI::CertificateRequestInfoList supportedCertRequestTypes;
        public PKI::CertificateRevocationInfoList supportedCertRevocationTypes;
        public PKI::KeyRecoveryInfoList supportedKeyRecoveryTypes;
        public PKI::Certificate publicKey;
        public string providerHomeURL;
        public string providerPublicKeyURL;
};

        exception UnsupportedTypeException {
                string description;
        };

        exception UnsupportedEncodingException {
                string description;
        };

        exception MalformedDataException {
                string description;
        };

        exception UnexpectedContinueException {
                string description;
        };


        interface RegistrationAuthority {

        AuthorityProviderInfo get_provider_info();

        PKI::PKIStatus get_authority_info(
                in PKI::AuthorityInfo authority_info_req,
                out PKI::AuthorityInfo authority_info_resp
                )
                raises(UnsupportedTypeException,UnsupportedEncodingException,
                                MalformedDataException);

        RequestCertificateManager request_certificate
                (in PKI::CertificateRequest certificate_request)
                raises(UnsupportedTypeException,UnsupportedEncodingException,
                                MalformedDataException);
```

```
RequestRevocationManager request_revocation
      (in PKI::CertRevRequest      cert_rev_request)
      raises(UnsupportedTypeException,UnsupportedEncodingException,
                  MalformedDataException);

RequestKeyUpdateManager request_key_update
      (in PKI::CertificateRequest   key_request)
      raises(UnsupportedTypeException,UnsupportedEncodingException,
                  MalformedDataException);

RequestKeyRecoveryManager request_key_recovery
      (in PKI::CertificateRequest   key_request)
      raises(UnsupportedTypeException,UnsupportedEncodingException,
                  MalformedDataException);
};

interface CertificateAuthority : RegistrationAuthority {

PKI::PKIStatus get_ca_certificate(
      out PKI::CertificateList certificate_list)
      raises (UnsupportedTypeException,UnsupportedEncodingException,
                  MalformedDataException);

PKI::PKIStatus get_crl (out PKI::CRL crl);

CertificateStatusResponder get_certificate_status_responder();

PKIRepository::Repository get_repository()
      raises(PKIRepository::RepositoryError);
};


abstract interface RequestManager {

readonly attribute PKI::PKIStatus status;

readonly attribute long transaction_ID;

void confirm_content(in PKI::ConfirmData     confirm_data)
      raises (UnsupportedTypeException,UnsupportedEncodingException,
                  MalformedDataException);
};

interface RequestCertificateManager : RequestManager {

void continue_request_certificate
      (in PKI::RequestData              request_data,
      in PKI::CertificateList           certificates)
      raises (UnsupportedTypeException,UnsupportedEncodingException,
                  MalformedDataException);

PKI::PKIStatus get_certificate_request_result
      (out  PKI::CertificateList        certificates,
      out PKI::ResponseData             response_data)
      raises (UnsupportedTypeException,UnsupportedEncodingException,
            MalformedDataException);
    };

    interface RequestRevocationManager : RequestManager {

    void continue_request_revocation
        (in PKI::RequestData            request_data)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
                MalformedDataException);

    PKI::PKIStatus get_request_revocation_result
        (out PKI::CertRevResponse   cert_rev_response,
        out PKI::ResponseData        response_data)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
                MalformedDataException);
    };

    interface RequestKeyUpdateManager : RequestManager {

    void continue_key_update
        (in PKI::RequestData      request_data,
        in PKI::Certificate        certificate)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
                MalformedDataException);

    PKI::PKIStatus get_request_key_update_result
        (out PKI::Certificate            certificate,
        out PKI::ResponseData        response_data)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
                MalformedDataException);
    };

    interface RequestKeyRecoveryManager : RequestManager {

    void continue_key_recovery
        (in PKI::RequestData            request_data)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
                MalformedDataException);

    PKI::PKIStatus get_request_key_recovery_result
        (out PKI::ResponseData       response_data)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
                MalformedDataException);
    };


    interface CertificateStatusResponder {

    PKI::PKIStatus request_certificate_status(
        in PKI::CertificateStatusRequest request,
        out PKI::CertificateStatusResponse response)
        raises (UnsupportedTypeException,UnsupportedEncodingException,
                MalformedDataException);
    };
};
```

```
#endif
```

A.3 PKIRepository

```
// PKIRepository.idl
#ifndef __PKIREPOSITORY_IDL
#define __PKIREPOSITORY_IDL

#include <PKI.idl>
#pragma prefix "omg.org"


module PKIRepository {

    valuetype RepositoryProviderInfo {
        public string standardDescription;
        public string standardVersion;
        public string productDescription;
        public string productVersion;
        public string productVendor;
        public PKI::CertificateInfoList supportedCertificates;
        public PKI::CRLInfoList supportedCRLs;
        public PKI::CertificateInfoList supportedCrossCertificates;
    };

    exception UnknownPrincipal {
        string name;
    };
    exception RepositoryError {
        string name;
    };
    exception DuplicatePrincipal {
        string name;
    };

    valuetype PKIPrincipalValue {
        private string name;
        private PKI::CertificateList certificates;
        private PKI::CertificatePairList;
        private PKI::CRL crl;
        private PKI::CRL delta;
        private PKI::CRL arl;
    };

    interface Repository {

    readonly attribute RepositoryProviderInfo info;

    void publish( in PKIPrincipalValue principal )
        raises ( DuplicatePrincipal, RepositoryError );

    PKIPrincipalValue locate ( in string name )
        raises ( UnknownPrincipal, RepositoryError );

    void delete ( in string name )
```

```
        raises ( UnknownPrincipal, RepositoryError );

    void update ( in PKIPrincipalValue principal )
        raises ( UnknownPrincipal, RepositoryError);
    };
};
#endif
```

A.4 PKIExtension

```
// PKIExtension.idl
#include<PKI.idl>
#include<PKIRepository.idl>

module PKIExtension {

    valuetype RepositoryMappingInfo {
    public string user_attribute_name;
    public string ca_attribute_name;
    public string crl_attribute_name;
    public string certificatePair_attribute_name;
    public string deltaCRL_attribute_name;
    public string arl_attribute_name;
    };

    typedef string PKIName;
    typedef sequence <PKIName> PKINameList;

    struct PKIAttribute {
        string name;
        any value;
    };
    typedef sequence <PKIAttribute> PKIAttributeList;

    struct PKIPrincipal {
        PKIName name;
        PKIAttributeList attributes;
    };

    struct Schema {
        PKIAttributeList attribute_defs;
        PKIAttributeList syntax_defs;
    };

    enum PrincipalAttributeErrorReason {
        MissingPKIAttributes,
        InvalidPKIAttributes
    };

    exception PrincipalAttributeError {
        PrincipalAttributeErrorReason reason;
        PKIPrincipal principal;
        PKINameList attribute_names;
    };
```

```
// renamed to LDAPRepository
interface LDAPRepository : PKIRepository::Repository {

// New method
RepositoryMappingInfo mapping();

Schema get_schema();

void publish_certificate(
      in PKIPrincipal principal,
      in PKI::Certificate certificate, in string attr_name)
      raises (PKIRepository::UnknownPrincipal,
      PrincipalAttributeError,
      PKIRepository::RepositoryError);

PKI::CertificateList get_certificate(
      in PKIPrincipal principal, in string attr_name)
      raises (PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

void delete_certificate(
      in PKIPrincipal principal,
      in PKI::Certificate certificate, in string attr_name)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

void publish_crl(in PKIPrincipal principal, in PKI::CRL crl,
      in string attr_name)
      raises(PKIRepository::UnknownPrincipal,
      PrincipalAttributeError,
      PKIRepository::RepositoryError);

PKI::CRL get_crl(in PKIPrincipal principal, in string attr_name)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

void delete_crl(in PKIPrincipal principal,
      in PKI::CRL crl,in string attr_name)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

void publish_certificate_pair(
      in PKIPrincipal principal, in PKI::CertificatePair certPair,
      in string attr_name)
      raises(PKIRepository::UnknownPrincipal,
      PrincipalAttributeError,
      PKIRepository::RepositoryError);

PKI::CertificatePairList get_certificate_pair(
      in PKIPrincipal principal, in string attr_name)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

void delete_certificate_pair(
      in PKIPrincipal principal,
      in PKI::CertificatePair certificate_pair,
      in string attr_name)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

void publish_user_certificate(in PKIPrincipal principal,
      in PKI::Certificate certificate)
      raises(PKIRepository::UnknownPrincipal,
      PrincipalAttributeError,
      PKIRepository::RepositoryError);

PKI::CertificateList get_user_certificate(in PKIPrincipal principal)
      raises(UnknownPrincipal,RepositoryError);

void delete_user_certificate(in PKIPrincipal principal,
      in PKI::Certificate certificate)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);


void publish_ca_certificate(
      in PKIPrincipal principal,
      in PKI::Certificate certificate)
      raises(PKIRepository::UnknownPrincipal,
      PrincipalAttributeError,
      PKIRepository::RepositoryError);

PKI::CertificateList get_ca_certificates(in PKIPrincipal principal)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

void delete_ca_certificate(in PKIPrincipal principal,
      in PKI::Certificate certificate)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

void publish_default_crl(in PKIPrincipal principal, in PKI::CRL crl)
      raises(PKIRepository::UnknownPrincipal,
      PrincipalAttributeError,
      PKIRepository::RepositoryError);

PKI::CRL get_default_crl(in PKIPrincipal principal)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

void delete_default_crl(in PKIPrincipal principal, in PKI::CRL crl)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

void publish_default_certificate_pair(in PKIPrincipal principal,
      in PKI::CertificatePair certificate_pair)
      raises(PKIRepository::UnknownPrincipal,
      PrincipalAttributeError,
      PKIRepository::RepositoryError);
      PKI::CertificatePairList get_default_certificate_pair(
          in PKIPrincipal principal)
          raises(PKIRepository::UnknownPrincipal,
          PKIRepository::RepositoryError);

      void delete_default_certificate_pair(in PKIPrincipal principal,
          in PKI::CertificatePair certificate_pair)
          raises(PKIRepository::UnknownPrincipal,
          PKIRepository::RepositoryError);

      void publish_delta_crl(in PKIPrincipal principal,
          in PKI::CRL delta_crl)
          raises(PKIRepository::UnknownPrincipal, PrincipalAttributeError,
          PKIRepository::RepositoryError);

      PKI::CRL get_delta_crl(in PKIPrincipal principal)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

      void delete_delta_crl(in PKIPrincipal principal, in PKI::CRL delta_crl)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

      void publish_arl(in PKIPrincipal principal, in PKI::CRL arl)
      raises(PKIRepository::UnknownPrincipal,
      PrincipalAttributeError,
      PKIRepository::RepositoryError);

      PKI::CRL get_arl(in PKIPrincipal principal)
      raises(PKIRepository::UnknownPrincipal,
      PKIRepository::RepositoryError);

      void delete_arl(in PKIPrincipal principal, in PKI::CRL arl)
      raises(PKIRepository::UnknownPrincipal,
      RPKIRepository::epositoryError);

      };
  };
```

Conformance Issues B

B.1 Introduction

This appendix specifies the conformance requirements to be met for an implementation to be conformant to the CORBA Public Key Infrastructure.

B.2 Conformance

There are 2 defined levels of conformance for the CORBA Public Key Infrastructure.

B.2.1 Level 1 : Polling Only

The first defined conformance level is for implementations that support polling only. For conformance to this level the following must be supported.

- Module PKIAuthority
  - Interface RegistrationAuthority
  - Interface CertificateAuthority
  - Interface RequestManager
  - Interface RequestCertificateManager
  - Interface RequestRevocationManager
  - Interface RequestKeyUpdateManager
  - Interface RequestkeyRecoveryManager
- Module PKIRepository
  - All specified constructs and interfaces

B.2.2 Level 2 : Polling and Callback

The second defined conformance level is for implementations that support both polling and callbacks. For conformance to this level the following must be supported.

- Module PKI
  - All specified constructs must be supported.
- Module PKIAuthority
  - Interface RegistrationAuthority_CB
  - Interface CertificateAuthority_CB
  - Interface RequestManager
  - Interface RequestCertificateManager
  - Interface RequestRevocationManager
  - Interface RequestKeyUpdateManager
  - Interface RequestkeyRecoveryManager
  - Interface CertificateCallback
  - Interface RevocationCallback
  - Interface KeyUpdateCallback
  - Interface KeyRecoveryCallback
- Module PKIRepository
  - All specified constructs and interfaces

Index

Public Key Infrastructure, v1.0
Reference Sheet

This is the first formal version of the Public Key Infrastructure specification.

OMG documents used to create this version:

- Submission document: ec/2000-02-01
- Convenience document: ptc/01-12-06