Public Key Infrastructure Specification

This OMG document replaces the submission document ec/2000-02-01. It is an OMG Final Adopted Specification, which has been approved by the OMG board and technical plenaries, and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by April 15, 2001.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*; however, at the time of this writing there were no pending issues.

The FTF Recommendation and Report for this specification will be published on July 18, 2001. If you are reading this after that date, please download the available specification from the OMG formal specifications web page.

OMG Adopted Specification

Public Key Infrastructure Specification

Final Adopted Specification
February 2001

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 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.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

OMG Documents

In addition to the CORBA Core Specification, OMG's document set includes the following publications.

OMG Modeling

The Unified Modeling Language (UML) Specification defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems. The specification includes the formal definition of a common Object Analysis and Design (OA&D) metamodel, a graphic notation, and a CORBA IDL facility that supports model interchange between OA&D tools and metadata repositories. The UML provides the foundation for specifying and sharing CORBA-based distributed object models.

The Meta-Object Facility (MOF) Specification defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models. The MOF provides the infrastructure for implementing CORBA-based design and reuse repositories. The MOF specifies precise mapping rules that enable the CORBA interfaces for metamodels to be automatically generated, thus encouraging consistency in manipulating metadata in all phases of the distributed application development cycle.

The OMG XML Metadata Interchange (XMI) Specification supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information. The specification supports the encoding of metadata consisting of both complete models and model fragments, as well as tool-specific extension metadata. XMI has optional support for interchange of metadata in differential form, and for metadata interchange with tools that have incomplete understanding of the metadata.

Object Management Architecture Guide

This document 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.

OMG Interface Definition Language (IDL) Mapping Specifications

These documents provide a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language. The OMG has an expanding set of language mappings, including Ada, C, C++, COBOL, IDL to Java, Java to IDL, Lisp, and Smalltalk.

CORBAservices

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal-application domain-independent basis for application interoperability.

These services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include Collection, Concurrency, Event, Externalization, Naming, Licensing, Life Cycle, Notification, Persistent Object, Property, Query, Relationship, Security, Time, Trader, and Transaction.

CORBAfacilities

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include Internationalization and Time, and Mobile Agent Facility.

Object Frameworks and Domain Interfaces

Unlike the interfaces to individual parts of the OMA “plumbing” infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Domain Task Forces concentrate on Object Framework specifications that include Domain Interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecoms, E-Commerce, and Transportation.

Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren’t required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to the *Common Object Request Broker: Architecture and Specification, Interworking Architecture* chapter.

Obtaining OMG Documents

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*.)

OMG formal (published) specifications are available from the OMG website *http://www.omg.org/technology/documents/formal/index.htm*. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted 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.

Source Document(s)

This FTF Adopted Specification is based on the following OMG document:

- ec/2000-02-01 - submission document

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.

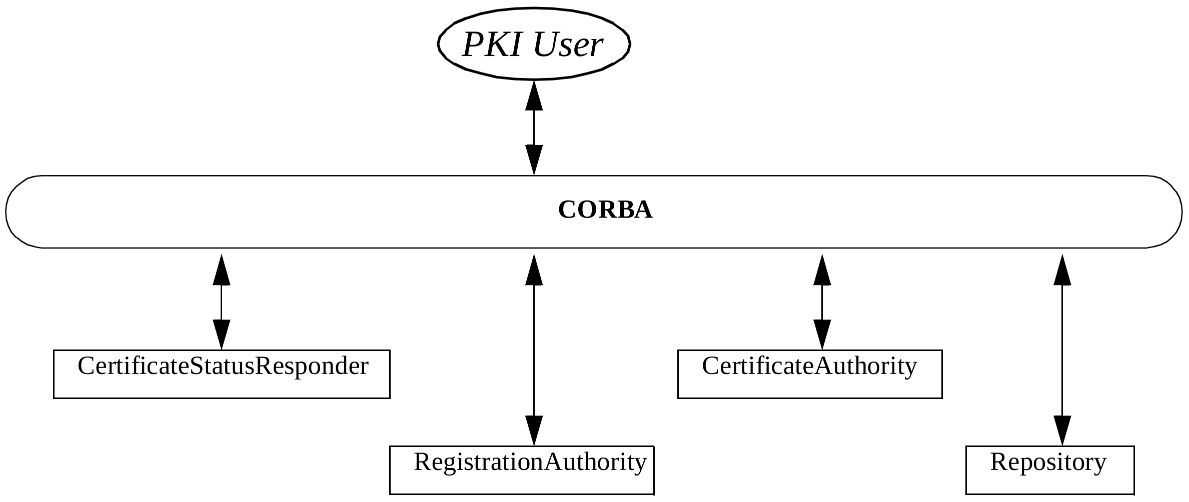


Figure 1-1 PKI Overview

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

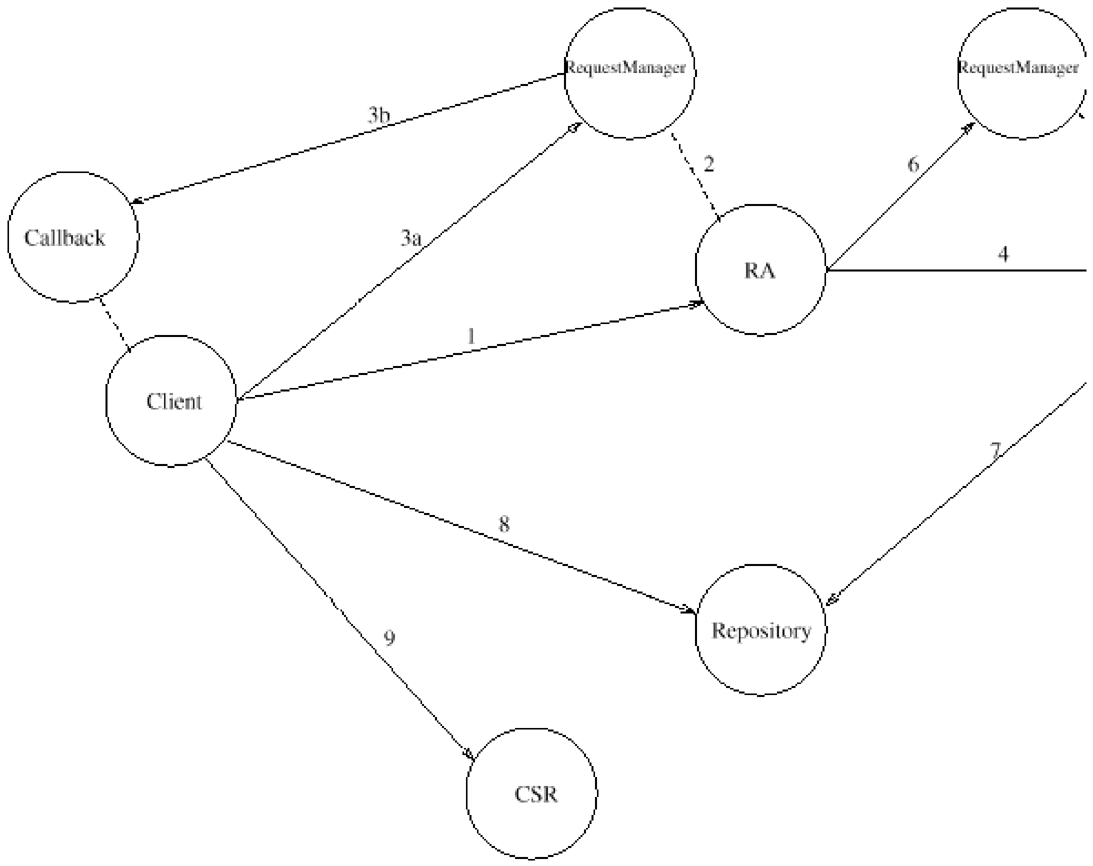
This section describes some typical usage scenarios for interaction with the interfaces described in this specification. As mentioned earlier there is the potential for communication between PKI users to have an interactive and asynchronous nature. The asynchronicity comes from the fact that an authority 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 asynchronous and interactive nature of PKI messaging additional interfaces have been added. These are the **RequestManager** and **Callback** interfaces. A **RequestManager** object is created by the authority. This is then used for any subsequent operations and status enquiries for a particular request. The callback interfaces are implemented at the client side and are used by the target to notify the client that a status change has occurred. This gives two possible methods for a client to interact with a CA or RA. The client can continuously poll the **RequestManager** object for a status change or it can register a local callback object and be notified of a status change. Whether an authority supports polling only or polling and callbacks will depend on the level of conformance as described in Appendix B - Conformance Issues.

Some example scenarios describing suggested behavior are overviewed in Section 1.4.3, “Polling Certificate Request,” on page 1-6 and Section 1.4.4, “Certificate Request Using A Callback,” on page 1-7.

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
3a) Client using polling to obtain results
3b) Client uses a Callback to be notified by target
4,5,6) RA performs client operations to make request on a 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 Polling 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 poll using either **status** or **get_certificate_request_result** operations and check the status until a change occurs. 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.

1.4.4 Certificate Request Using A Callback

This example uses a call back to request issuance of a certificate from an authority.

- After constructing a request, using a supported standard, the message can be encapsulated and invoked through the **request_certificate_with_CB** operation passing in a reference to a client side **CertificateCallback** object. A reference to a **RequestCertificateManager** object is also returned (as in the above example) to be used if more interaction is required for this request.
- The client can then wait (or perform other tasks) until the reply handler's **notify** operation is called by the target. Once notified the results can 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 poller object using the **continue_request_certificate** operation until a success (or failure) is reached, again using the callback object for notification from the target.

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 which is discussed in Section 5.6. 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. If this is to be performed in a single synchronous invocation the potential for lengthy delays may cause timeout problems. To address this, a method of handling asynchronicity was added in the form of **RequestManager** and **Callback** interfaces.

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. This implies a server side model for asynchronous messaging. Since the CORBA *Asynchronous Messaging Service* specification in principle provides a client side asynchronous model with no changes to the server side it does not specifically suit this particular domain. Also since the state and processing is on the server side it is logical for the callback reference to be known by the server side authority for notification of a status change. The **RequestManager** interface is created by the authority and then encapsulates everything that relates to this 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.

1.6.3 Repository

The repository was primarily designed to allow for implementations that use the X.500 or LDAP directory services, which is seen as being the predominant method of repository implementation in existing services. However, it may be the case that a PKI repository is implemented using some other data storage service, such as a database. In either case, the data storage service 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 *Opaque*

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

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

2.2.2 RepresentationType

```
struct RepresentationType {
    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.

2.2.3 Certificate

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

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.
representation_type	This is used to contain the actual representation of the certificate. For existing PKI standards and implementations this will be **RepresentationType** defined in this module. A local value object type may also be used here in representing a certificate in future revisions of this document.

2.2.4 CRL

This construct is the representation of a Certificate Revocation List.

```
struct CRL {
    CRLType crl_type;
    any representation_type;
};
```

Fields

crl_type	The type of CRL such as X509V1CRL, X509V2CRL, X509V1ARL.
representation_type	This is used to contain the actual representation of the CRL. For existing PKI standards and implementations this will be **RepresentationType** defined in this module. A local value object type may also be used here in representing a CRL in future revisions of this document.

2.2.5 CertificateRequest

The construct used to represent an encoded certificate request message.

```
struct CertificateRequest {
   CertificateRequestType cert_request_type;
   any representation_type;
};
```

Fields

cert_request_type	The type of certificate request message such as PKCS10, PKIXCRMF, PKIXCMC.
representation_type	This is used to contain the actual representation of the certificate request message. For existing PKI standards and implementations this will be **RepresentationType** defined in this module. A local value object type may also be used here in representing a certificate request in future revisions of this document.

2.2.6 CertificateStatusRequest

```
struct CertificateStatusRequest {
   CertificateStatusRequestType type;
   any value;
};
```

Fields

type	The type of certificate status request such as OCSP.
value	This is used to contain the actual representation of the certificate status request message. For existing PKI standards and implementations this will be **RepresentationType** defined in this module. A local value object type may also be used here in representing a certificate status request in future revisions of this document.

2.2.7 *CertificateStatusResponse*

```
struct CertificateStatusResponse {
   CertificateStatusResponseType type;
   any value;
};
```

Fields

type	The type of certificate status response such as OCSP.
value	This is used to contain the actual representation of the certificate status response message. For existing PKI standards and implementations this will be **RepresentationType** defined in this module. A local value object type may also be used here in representing a certificate status response in future revisions of this document.

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
        );
```

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);
```

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);
```

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);
```

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);
```

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 RegistrationAuthority_CB*

Interface defining operations that can be performed on a **RegistrationAuthority_CB** object. This interface is used where the implementation provides conformance to level 2, as described in Appendix B, Conformance Issues (i.e., supports both polling and callbacks). This interface inherits operations from **RegistrationAuthority**.

2.3.2.1 *request_certificate_with_CB*

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

```
RequestCertificateManager request_certificate_with_CB
        (in CertificateCallback callback,
        in PKI::CertificateRequest certificate_request) ;
```

Parameters

callback	The client's callback **CertificateCallback** object used by the target object to notify the client of a change in status of the request.
certificate_request	**PKI::CertificateRequest** structure containing details of the client's request.

Return Value

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

2.3.2.2 *request_revocation_with_CB*

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

```
 RequestRevocationManager request_revocation_with_CB
        (in RevocationCallback callback,
        in PKI::CertRevRequest     cert_rev_request);
```

Parameters

callback	The client's callback **RevocationCallback** object that is used by the target object to notify the client of a change in status of the request.
cert_rev_request	**PKI::CertRevRequest** structure containing details of the client's request.

Return Value

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

2.3.2.3 *request_key_update_with_CB*

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

```
RequestKeyUpdateManager request_key_update_with_CB
        (in KeyUpdateCallback callback,
        in PKI::CertificateRequest   key_request);
```

Parameters

callback	The client's callback **KeyUpdateCallback** object used by target object to notify the client of a change in status of the request.
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.

2.3.2.4 *request_key_recovery_with_CB*

Called to request key recovery from a (CA) or (RA) where the authority provides key archive using a callback object.

```
RequestKeyRecoveryManager request_key_recovery_with_CB
        (in KeyRecoveryCallback callback,
        in PKI::CertificateRequest   key_request) ;
```

Parameters

callback	The client's callback **KeyRecoveryCallback** object used by target object to notify the client of a change in status of the request and return results.
key_request	**PKI::CertificateRequest** structure containing details of the client's request for key recovery.

Return Value

RequestKeyRecoveryManager object reference used to extract details pertaining to the request and obtain results.

2.3.3 Interface CertificateAuthority

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

2.3.3.1 get_ca_certificate

Returns the certificates of the **CertificateAuthority**.

```
PKI::PKIStatus get_ca_certificate(
        out PKI::CertificateList certificate_list);
```

Parameters

certificate_list	List of certificates for CA.

Return Value

PKI::PKIStatus indicating the status of the request.

2.3.3.2 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.

2.3.3.3 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.

2.3.3.4 get_repository

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

PKIRepository::Repository get_repository();

Return Value

PKIRepository::Repository object reference.

2.3.4 Interface CertificateAuthority_CB

Interface defining operations that can be performed on a **CertificateAuthority_CB** object. This interface is used where the implementation provides conformance to level 2, as described in Appendix B, Conformance Issues (i.e., supports both polling and callbacks). This interface inherits operations from both **RegistrationAuthority_CB** and **CertificateAuthority**, and adds no new operations.

2.3.5 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.

2.3.5.1 status

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

readonly attribute PKI::PKIStatus status;

2.3.5.2 transaction_ID

A read only attribute representing an identifier for a particular transaction.

readonly attribute long transaction_ID;

2.3.5.3 confirm_content

Operation to acknowledge negotiation is complete.

void confirm_content(in PKI::ConfirmData   confirm_data);

Parameters

confirm_data	Message to confirm content is correct and received.

2.3.6 Interface RequestCertificateManager

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

2.3.6.1 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);
```

Parameters

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

2.3.6.2 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);
```

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.

2.3.7 Interface RequestRevocationManager

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

2.3.7.1 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);
```

Parameters

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

2.3.7.2 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);
```

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.

2.3.8 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.

2.3.8.1 *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);
```

Parameters

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

2.3.8.2 *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) ;
```

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.

2.3.9 *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.

2.3.9.1 *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);
```

Parameters

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

2.3.9.2 *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);
```

Parameters

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

Return Value

PKI::PKIStatus indicating the status of the request.

2.3.10 *Interface CertificateCallback*

Callback interface implemented at client side for being notified of a result.

2.3.10.1 *notify_result*

```
void notify_result(
                in RequestCertificateManager req_cert_manager,
                in PKI::PKIStatus status,
                in PKI::CertificateList   certificates,
                in PKI::ResponseData response_data) ;
```

Parameters

req_cert_manager	An object reference to a **RequestCertificateManager** object that maintains details of a certificate request.
status	**PKI::Status** indicating current status of request.
certificates	A list of certificates.
response_data	**PKI::ResponseData** structure that holds details of the server side response for the request.

2.3.11 *Interface RevocationCallback*

Callback interface implemented at client side for being notified of a result.

2.3.11.1 *notify_result*

Used by the server side to notify the reply handler that there is a result, either interim or final.

```
void notify_result(in RequestRevocationManager req_rev_manager,
                   in PKI::PKIStatus status,
                   in PKI::ResponseData response_data) ;
```

Parameters

req_rev_manager	An object reference to a **RequestRevocationManager** object that maintains details of a request revocation request.
status	**PKI::Status** indicating current status of request.
response_data	**PKI::ResponseData** structure that holds details of the server side response for the request.

2.3.12 *Interface KeyUpdateCallback*

Callback interface implemented at client side for being notified of a result.

2.3.12.1 *notify_result*

Used by the server side to notify the reply handler that there is a result, either interim or final.

```
void notify_result(in RequestKeyUpdateManager req_key_update_manager,
                   in PKI::PKIStatus status,
                   in PKI::ResponseData response_data) ;
```

Parameters

req_key_update_manager	An object reference to a **RequestKeyUpdateManager** object that maintains details of a key update request.
status	**PKI::Status** indicating current status of request.
response_data	**PKI::ResponseData** structure that holds details of the server side response for the request.

2.3.13 *Interface KeyRecoveryCallback*

Callback interface implemented at client side for being notified of a result.

2.3.13.1 *notify_result*

Used by the server side to notify the reply handler that there is a result, either interim or final.

```
void notify_result
        (in RequestKeyRecoveryManage req_key_recover_manager,
                in PKI::PKIStatus status,
                in PKI::ResponseData response_data) ;
```

Parameters

req_key_recover_manager	An object reference to a **RequestKeyRecoveryManager** object that maintains details of a key recovery request.
status	**PKI::Status** indicating current status of request.
response_data	**PKI::ResponseData** structure that holds details of the server side response for the request.

2.3.14 *Interface CertificateStatusResponder*

Interface for an online certificate status responder.

2.3.14.1 *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);
```

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 Interface Repository

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

2.4.1.1 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.

2.4.1.2 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.

2.4.1.3 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);
```

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.

2.4.1.4 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);
```

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.

2.4.1.5 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);
```

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.

2.4.1.6 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);
```

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.

2.4.1.7 *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);

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.

2.4.1.8 *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)
```

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.

2.4.1.9 *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);

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.

2.4.1.10 *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);

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.

2.4.1.11 *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);

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.

2.4.1.12 *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);
```

Parameters

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

2.4.1.13 *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);
```

Parameters

principal	The principal to which the certificate is bound.

Return Value

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

2.4.1.14 *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)  ;
```

Parameters

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

2.4.1.15 *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);
```

Parameters

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

2.4.1.16 *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);
```

Parameters

principal	The principal to which the CA certificates are bound.

Return Value

PKI::CertificateList containing requested CA certificates.

2.4.1.17 *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)
```

Parameters

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

2.4.1.18 *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);

Parameters

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

2.4.1.19 *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);

Parameters

principal	The principal to which the CRL is bound.

Return Value

The PKI::CRL containing the requested CRL.

2.4.1.20 *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);

Parameters

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

2.4.1.21 *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)
```

Parameters

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

2.4.1.22 *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);
```

Parameters

principal	The principal to which the certificate pairs are bound.

Return Value

PKI::CertificatePairList containing requested certificate pairs.

2.4.1.23 *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)  ;
```

Parameters

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

2.4.1.24 *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.

2.4.1.25 *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);

Parameters

principal	The principal to which the delta CRL is bound.

Return Value

PKI::CRL containing the requested delta CRL.

2.4.1.26 *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);

Parameters

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

2.4.1.27 *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);

Parameters

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

2.4.1.28 *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);

Parameters

principal	The principal to which the ARL is bound.

Return Value

PKI::CRL containing the requested ARL.

2.4.1.29 *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);

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


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;

//CRL
struct CRL {
    CRLType crl_type;
    any representation_type;
};



//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 {
    CertificateRequestType cert_request_type;
    any representation_type;
};

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
  struct Continue {
     ContinueType continue_type;
     any representation_type;
  };

  //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
  struct CertRevocation {
     CertRevocationType cert_rev_type;
     any representation_type;
  };

  //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
  struct KeyRecResponse {
     KeyRecoveryType key_recovery;
     any representation_type;
  };

  //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
  struct CertificateStatusRequest {
     CertificateStatusRequestType type;
     any value;
  };

  //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
  struct CertificateStatusResponse {
     CertificateStatusResponseType type;
     any value;
  };

  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;

    interface CertificateCallback;
    interface RevocationCallback;
    interface KeyUpdateCallback;
    interface KeyRecoveryCallback;

    struct AuthorityProviderInfo {
        string standardVersion;
        string standardDescription;
        string productVersion;
        string productDescription;
        string productVendor;
        PKI::CertificateInfoList supportedCertificates;
        PKI::CRLInfoList supportedCRLs;
        PKI::CertificateRequestInfoList supportedCertRequestTypes;
        PKI::CertificateRevocationInfo supportedCertRevocationTypes;
        PKI::KeyRecoveryInfoList supportedKeyRecoveryTypes;
        boolean callbackSupport;
    };

    exception UnsupportedTypeException {
        string description;
    };

    exception UnsupportedEncodingException {
        string description;
    };

    exception MalformedDataException {
```

```
    string description;
};

exception UnexpectedContinueException {
     string description;
};

exception InvalidCallbackException {
    string description;
};

interface RegistrationAuthority {

    AuthorityProviderInfo get_provider_info();

    PKI::PKIStatus get_authority_info(
        in PKI::AuthorityInfo  in_authority_info,
        out PKI::AuthorityInfo out_authority_info
        )
        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 RegistrationAuthority_CB : RegistrationAuthority
    {

    RequestCertificateManager request_certificate_with_CB
        (in CertificateCallback callback,
        in PKI::CertificateRequest certificate_request)
        raises(UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
        RequestRevocationManager request_revocation_with_CB
            (in RevocationCallback callback,
            in PKI::CertRevRequest      cert_rev_request)
            raises(UnsupportedTypeException,UnsupportedEncodingException,
                  MalformedDataException);

        RequestKeyUpdateManager request_key_update_with_CB
            (in KeyUpdateCallback callback,
            in PKI::CertificateRequest   key_request)
            raises(UnsupportedTypeException,UnsupportedEncodingException,
                  MalformedDataException);

        RequestKeyRecoveryManager request_key_recovery_with_CB
            (in KeyRecoveryCallback callback,
            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);
    };

    interface CertificateAuthority_CB : RegistrationAuthority_CB,
                                CertificateAuthority {
    };

    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 CertificateCallback {

     void notify_result(in RequestCertificateManager req_cert_manager,
                    in PKI::PKIStatus status,
                    in PKI::CertificateList    certificates,
                    in PKI::ResponseData response_data)
       raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);

  };

  interface RevocationCallback {

     void notify_result(in RequestRevocationManager req_rev_manager,
                    in PKI::PKIStatus status,
                    in PKI::ResponseData response_data)
       raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
  };

  interface KeyUpdateCallback {

     void notify_result(in RequestKeyUpdateManager
req_key_update_manager,
                    in PKI::PKIStatus status,
                    in PKI::ResponseData response_data)
       raises (UnsupportedTypeException,UnsupportedEncodingException,
              MalformedDataException);
  };

  interface KeyRecoveryCallback {


     void notify_result(in RequestKeyRecoveryManager
req_key_recover_manager,
                    in PKI::PKIStatus status,
                    in 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

```
#define __PKIREPOSITORY_IDL

#include <PKI.idl>

#pragma prefix “omg.org”

module PKIRepository{

   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;
   };

   struct RepositoryProviderInfo {
      string standardDescription;
      string standardVersion;
      string productDescription;
      string productVersion;
      string productVendor;
      PKI::CertificateInfoList supportedCertificates;
      PKI::CRLInfoList supportedCRLs;
      PKI::CertificateInfoList supportedCrossCertificates;
      string user_attribute_name;
      string ca_attribute_name;
      string crl_attribute_name;
      string certificatePair_attribute_name;
      string deltaCRL_attribute_name;
```

```
    string arl_attribute_name;
};

exception UnkownPrincipal {
    PKIPrincipal principal;
};

enum PrincipalAttributeErrorReason {
    MissingPKIAttributes,
    InvalidPKIAttributes
};

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

exception RepositoryError {
    string reason;
};

interface Repository {

    RepositoryProviderInfo get_provider_info();

    Schema get_schema();

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

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

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

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

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

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

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

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

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

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

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

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

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

    PKI::CertificateList get_ca_certificate(in PKIPrincipal principal)
        raises(UnkownPrincipal,RepositoryError);

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

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

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

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

    void publish_default_certificate_pair(in PKIPrincipal principal,
                                  in PKI::CertificatePair certificate_pair)
```

```
                raises(UnkownPrincipal,PrincipalAttributeError,RepositoryError);

            PKI::CertificatePairList get_default_certificate_pair(
                in PKIPrincipal principal)
                raises(UnkownPrincipal,RepositoryError);

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

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

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

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

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

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

            void delete_arl(in PKIPrincipal principal, in PKI::CRL arl)
              raises(UnkownPrincipal,RepositoryError);

        };
    };
    #endif
```

Conformance Issues B

B.1 Introduction

This chapter 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 PKI
  - All specified constructs
- 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