Defence Research and Development Canada
Recherche et développement pour la défense Canada

Policy Decision Point (PDP) Software Design Document

J. Spagnolo, D. Cayer

Defence R&D Canada – Ottawa
CONTRACT REPORT
DRDC Ottawa CR 2005-110
September 2005

Canada

Policy Decision Point (PDP) Software Design Document

J. Spagnolo, D. Cayer
NRNS Incorporated

NRNS Incorporated
4043 Carling Avenue
Ottawa, ON
K2K 2A3

Contract Number: W7714-3-800/001/SV

Contract Scientific Authority: Dr. S. Zeber (613) 991-1388

Contract Scientific Advisor: Mr. Tim Symchych, CRC, (613) 949 - 3070

The work described in this report was sponsored jointly by the Department of National Defence under the work unit 15BF and by the Communications Research Center Canada under the work units 15CS and 15CV.

Defence R&D Canada – Ottawa

Contract Report

DRDC Ottawa CR 2005-110

September 2005

Terms of release: The scientific or technical validity of this Contract Report is entirely the responsibility of the contractor and the contents do not necessarily have the approval or endorsement of Defence R&D Canada or the Communications Research Centre.

Document Revision History

Version 0.1	31-Mar-2005	Complete DRAFT submitted Scientific Authority for review.
Version 1.0	31-Mar-2005	Incorporated comments from Defence R&D Canada and the Communications Research Centre.
Version 1.1	12-Sept-2005	Added section for Policy Server.

Table of Contents

List of Figures

1. Introduction

Policy Based Network Management (PBNM) systems provide an automated means to configure and administer Policy Enforcement Point (PEP) devices such as virtual private network (VPN) gateways, firewalls and routers. The Policy Decision Point (PDP) takes high level policies as input and produces lower level PEP specific policies as output. The PBNM system can process different types of policies. When evaluating policies, the PDP must identify and resolve conflicts within competing policies as well as take into consideration external factors such as the time-of-day and the current threat level.

A PBNM system alleviates the need for network administrators to manually configure numerous network devices in order to implement local policy changes. We also introduce the concept of policy negotiation for inter-domain policies[1] such as inter-domain security policies. Negotiable policies are not complete policy documents and therefore the PDP cannot directly implement them. Instead, the local PDP must exchange policy proposals with a PDP in a remote administrative domain. Policy proposals contain all the negotiation parameters needed by the other party to correctly evaluate the proposed policy against the local policy. A PDP can accept a proposed policy in whole or in part or it can reject the proposed policy. If both parties accept the other party's proposed policy in whole or in part, each party merges the local and remote policy proposals to form a complete policy document that the PDP can implement. The PBNM system automatically reconfigures network devices as required to implement negotiated policies.

2. Purpose

This document presents a software design for the Policy Decision Point (PDP) component of PBNM system described in the "Policy Based Network Management - System Design Document" [PBNM]. The system facilitates the compilation of policies, the storage of policies, the exchange of policies, the evaluation and negotiation of policies, as well as the implementation and enforcement of policies.

The PDP is implemented in Java. The primary goal of this software design is to create an extensible PDP framework that can support different types of policies without requiring significant modifications to the existing software. Moreover, this software design makes extensive use of Java software interfaces to define the functions of the major system components. This allows for different implementations of the same software component to be substituted into the system without affecting the remainder of the system.

3. Software Architecture

Figure 1 illustrates major software components that comprise software architecture for the PDP. Most software components are defined by a Java interface that describes their intended purpose. This hides the implementation details and allows other objects to interact with the software component using the generic Java interface.

[1] Note that inter-domain policies may also be intra-organizational policies as some "domains" will be identified parts of larger organizations.

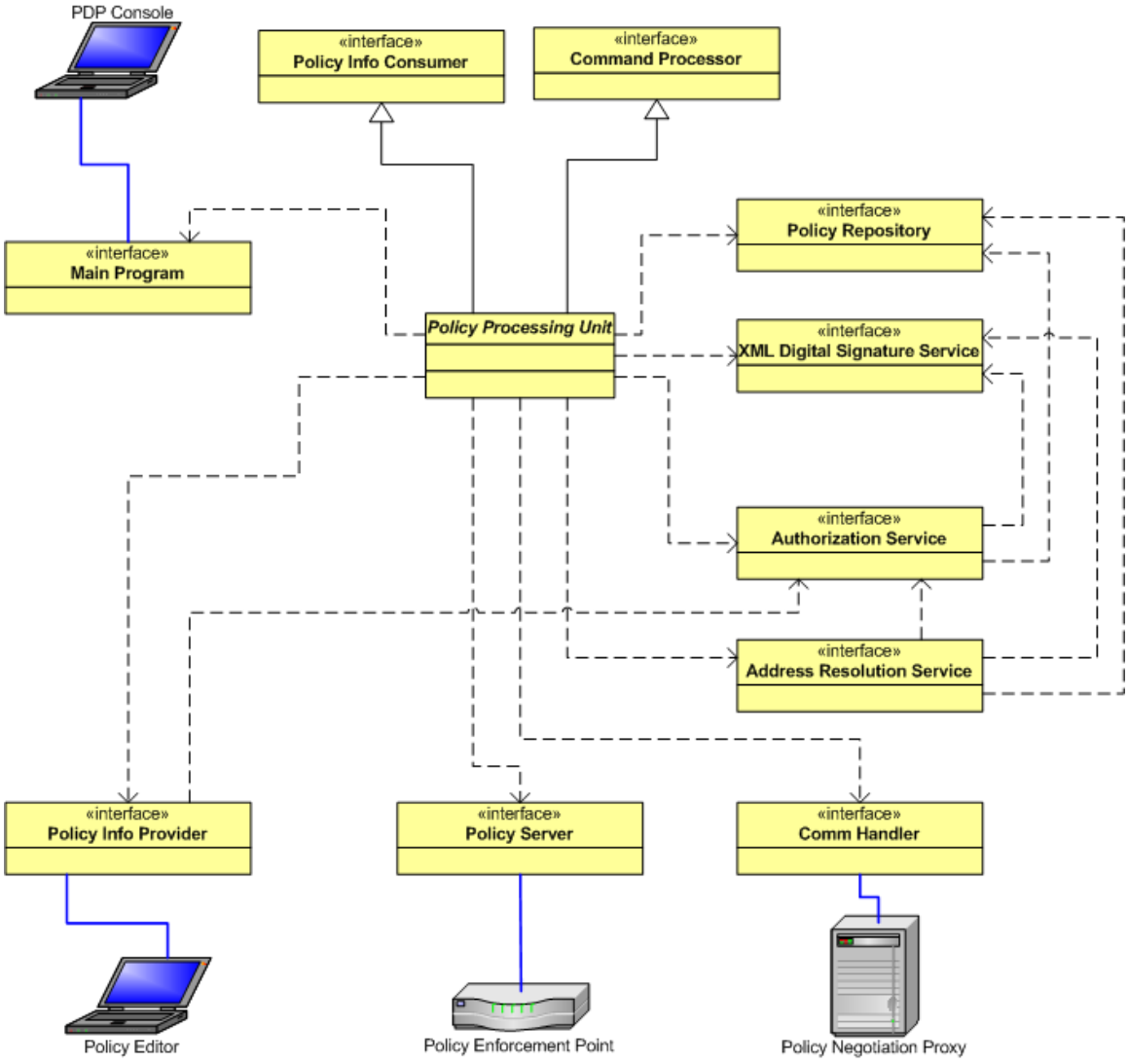

Figure 1 - PDP High Level Software Architecture

The *Policy Processing Unit* (PPU) forms the core of the PDP and is the only component that possesses knowledge of policy structure, syntax and encoding. The PDP requires the implementation of a distinct *PPU* class for each type of policy supported by the system. Section **Error! Reference source not found.**, which describes the *PPU* class in more detail, describes how a specific *PPU* class is implemented by extending a pair of abstract Java classes.

The *PPU* class[2] implements two interfaces. The *Policy Info Consumer* interface allows the *PPU* to acquire new policy documents from the Policy Editor via the *Policy Info Provider*, while the *Command Processor* interface allows the *PPU* to accept user commands from the PDP Console via the *Main Program*.

The *PPU* makes extensive use of the four infrastructure services. The *Policy Repository* provides access to an XML based directory where the *PPU* stores policy objects. The *XML Digital*

[2] In reality, PPU inner classes implement these interfaces.

Signature Service applies digital signatures to XML documents and verifies digital signatures on XML documents. The *Authorization Service* determines which operations an entity (based on the entity's digital credentials) can perform. The *Address Resolution Service* maps between administrative domain names and the associated network layer information. In addition to the *PPU*, other PDP software components make use of these infrastructure services, including the infrastructure services themselves.

The *Policy Info Provider* interacts with the Policy Editor to acquire new policy documents and submits them to interested consumers. Currently only *PPU* objects act as *Policy Info Consumers*. In the future however, other PDP objects will accept policy documents from the Policy Editor and they will also implement the *Policy Info Consumer* interface.

For inter-domain negotiable policies, the *Comm Handler* facilitates communication between the various *PPU* objects within the PDP system and the Policy Negotiation Proxy (PNP) system. The PBNM system supports the simultaneous negotiation of different types of policies.

The *Policy Server* accepts device level policy documents from the various *PPU* objects to be implemented within Policy Enforcement Point (PEP) devices. The *Policy Server* accepts Common Object Policy Service (COPS) for provisioning (COPS-PR) sessions from PEP devices and supplies the relevant portions of device level policy documents as configuration information to PEP devices based on the PEP device's type or role.

The *Main Program* provides the entry point for the PDP system. It instantiates all the major software components shown in Figure 1 and provides methods that allow PDP software components to acquire references to other PDP software components. This is how the *PPU* objects acquire references to the objects that provide infrastructure services, as well as to the *Policy Info Provider*, the *Comm Handler*, and the *Policy Server*. The *Main Program* also provides a basic command line interface for the PDP system through the PDP Console.

Figure 1 does not show the *Status Code Manager*, which maps PBNM status codes to informative strings. Like the other major software components, the *Status Code Manager* is also a Java interface that can realize many different implementations. Internationalization can be easily achieved by creating different implementations of the *Status Code Manager* interface.

4. Compliance with System Design

The software design described in this document implements the PDP portion of the PBNM system described in [PBNM] – with the following exceptions:

1. The PDP only includes support for negotiable inter-domain policies. Static policies are not yet supported.

2. The communication between the PDP and the PNP as well as the inter-PNP communication is implemented using JGroups [JGROUPS].

3. Due to limitations with JGroups, the PDP/PNP communication channel and the inter-PNP communication channels are not authenticated or secured. The system is easily extensible however and could easily incorporate different communication frameworks (i.e. Transport Layer Security, Session Initiation Protocol).

4. The Policy Editor does not possess the ability to retrieve a policy document from the PDP. As such, policy documents are stored locally on the system housing the Policy Editor.

5. The PDP does not preserve the digital signature of the policy document creator when it applies the PDP digital signature on a newly submitted policy document. The policy documents stored in the directory only contain the PDP's single digital signature.

6. The PDP system does not currently perform status checks on certificates used to authenticate communication channels and to sign policy objects.

7. The PDP does not currently retrieve policy negotiation artifacts at start-up.

5. Software Design

This section presents a high-level design for each PDP software component. It provides an overview of each component, it lists the requirements for each component, and it outlines the design for each component in terms of processing, thread synchronization and exceptions.

5.1 Base PBNM Objects

This section provides a description of some of the base PBNM software objects referenced in the remainder of this document.

AlertListener	An AlertListener is an interface that a class implements when it wants to be alerted of asynchronous events from objects executing in a different thread. The alert() method accepts a single object of type Object as an argument.
Reminder	A Reminder is an object that dispatches an alert at a specific time in the future. The recipient of the alert must be an AlertListener.
NetworkID	A NetworkID contains the information that is needed to communicate with another network entity. This includes network layer (IPv4 or IPv6) addresses and transport port numbers.
DigitalID	A DigitalID contains a generic certificate that uniquely identifies an entity – an individual, system or process. The system currently supports digital identifiers based on X.509 certificates.
PBNMKeyStore	A PBNMKeyStore contains generic digital credentials that include both a private key and a public certificate. The system currently supports digital credentials enclosed within PKCS #12 files.
SynchronizedQueue	A SynchronizedQueue allows two threads to exchange objects/messages in a synchronized fashion.

PolicyInfo	A PolicyInfo object carries a policy document between the Policy Editor and the PPU responsible for its processing. A PolicyInfo object includes a header that identifies the type of policy and the type of policy object contained within its payload.
PNU	A policy negotiation unit (PNU) carries policy negotiation objects between the PNP and the PPU responsible for its processing. A PNU includes a header that identifies the type of policy and the type of policy object contained within its payload.
Read/Write Lock	A read/write lock synchronizes access to objects – allowing access by single writer when no readers are active or by many concurrent readers when no writers are active.
LockableDocument	A LockableDocument adds a read/write lock to the org.w3c.dom.Document class. Callers apply and remove read or write locks on the Document prior to accessing it.
LockableHashtable	A LockableHashtable adds a read/write lock to the java.util.Hashtable class. Callers apply and remove read or write locks on the Hashtable prior to accessing it.

5.2 Policy Repository

The *Policy Repository* stores and retrieves XML documents from a XML database.

Figure 2 shows the UML (Unified Modelling Language) class diagram for the *Policy Repository*. The *XindiceRepository* class implements the *PolicyRepository* interface, which mandates the implementation of the following public methods:
Figure 2 shows the UML (Unified Modelling Language) class diagram for the *Policy Repository*. The *XindiceRepository* class implements the *PolicyRepository* interface, which mandates the implementation of the following public methods:

retrieve()	Retrieve a specific document from the database.
store()	Store a specific XML document in the database.

The retrieve() and store() methods execute in the context of the caller's thread. They perform their intended functions with the use of blocking I/O operations and as such they will cause the caller's thread to block until the I/O operation completes.

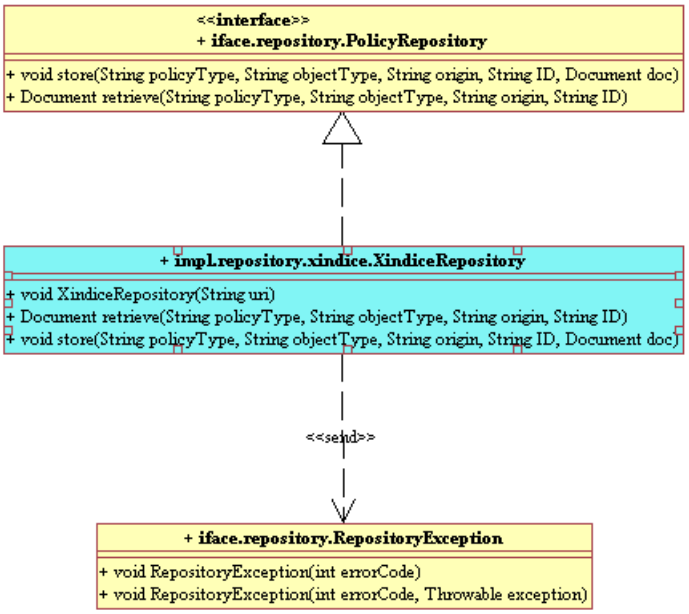

Figure 2 - Policy Repository UML Diagram

5.2.1 Requirements

The *XindiceRepository* must fulfill the following requirements:

1. Register with the database server.
2. Retrieve a specific document from the database.
3. Store a specific XML document in the database.

5.2.2 Design

The *Main Program* creates a single instance of the *XindiceRepository* class. The lone *XindiceRepository* constructor takes a single argument – the Uniform Resource Identifier (URI) of the Apache Xindice database server. Once the *XindiceRepository* constructor registers with the database server, the *XindiceRepository* is ready to perform retrieve and store operations.

5.2.2.1 Synchronization

The *XindiceRepository* class does not concern itself with thread synchronization.

5.2.2.2 Exceptions

The *XindiceRepository* class throws a *RepositoryException* exception if it encounters difficulties registering with the database server, or if a retrieve or store operation results in an error condition.

5.3 XML Digital Signature Service

The *XML Digital Signature Service* provides a XML based digital signature service to other PDP software components.

Figure 3 shows the UML class diagram for the *XML Digital Signature Service*. The *Jsr105XMLDsigService* class implements the *XMLDsigService* interface, which mandates the implementation of the following public methods:

sign()	Apply a signature to the supplied XML document.
verify()	Verify the outer most digital signature on the supplied XML document and optionally remove the digital signature element from the XML document.

The sign() and verify() methods execute in the context of the caller's thread. They perform their intended functions without the use of blocking I/O operations and as such they will not cause the caller's thread to block for an extended period of time.

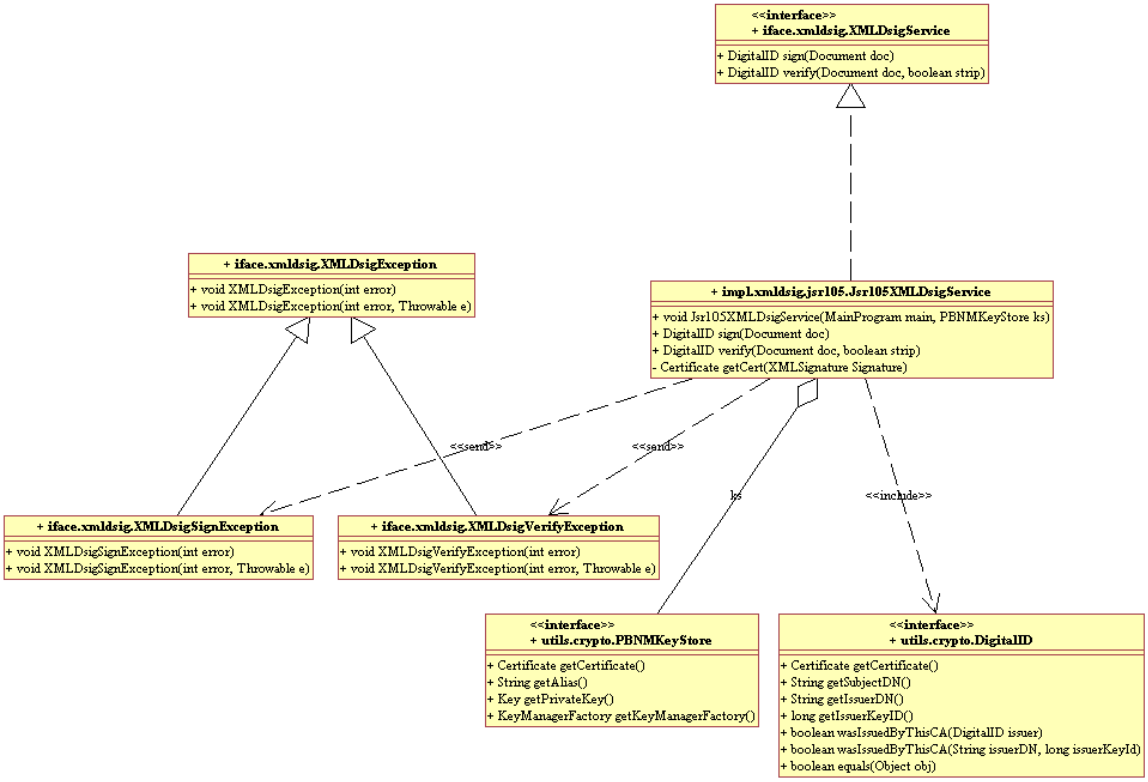

Figure 3 - XML Digital Signature Service Interface

5.3.1 Requirements

The *Jsr105XMLDsigService* must fulfill the following requirements:

1. Accept and manage the digital credentials, including the private key.

2. Verify the outer most digital signature on an XML document and optionally remove the digital signature element from the XML document.

3. Apply a digital signature to a XML document.

In the future, the *XML Digital Signature Service* should also accept new digital credentials at run-time.

5.3.2 Design

The *Main Program* creates a single instance of the *Jsr105XMLDsigService* class. The lone *Jsr105XMLDsigService* constructor takes two arguments: a reference to the *Main Program*, and a reference to a *PBNMKeyStore* that holds the digital credentials of the local PDP. The constructor stores the reference to the *PBNMKeyStore* for use by the public sign() method.

The *Jsr105XMLDsigService* is ready to provide XML based digital signature services once the constructor completes. Both the sign() and verify() methods take an XML document as an argument. The sign() and verify methods manipulate the document object directly – no copies are made. Both the sign() and verify methods return a *DigitalID*, which contains the digital credentials (certificate) of the entity that signed the XML document.

5.3.2.1 Synchronization

The *Jsr105XMLDsigService* class uses class level synchronization to ensure that only a single instance of the class is active within the system.

The public sign() and verify() methods are fully synchronized although they do not try to alter instance variables. The absence of method synchronization resulted in signature validation failure during concurrent load testing. Additional research is needed to determine why these methods must be synchronized.

In the future, the *Jsr105XMLDsigService* class may accept new digital credentials at run-time. In case method synchronization is removed in the future, the public sign() method copies the reference to the current digital credentials into a local variable to ensure that the private key used to sign the document matches the public certificate inserted into the document.

5.3.2.2 Exceptions

The *Jsr105XMLDsigService* class throws the *XMLDsigException* when an attempt to instantiate more than one instance of the class is attempted.

The *Jsr105XMLDsigService* class throws the *XMLDsigSignException* if it encounters difficulties in applying a digital signature to the XML document.

The *Jsr105XMLDsigService* class throws the *XMLDsigVerifyException* if it encounters difficulties verifying the digital signature on the supplied XML document.

5.4 Authorization Service

The *Authorization Service* determines whether or not a specific entity is authorized to perform a specific activity. Examples of the type of actions supported by the Simple Authorization Service include:

- Who can establish an SSL session using the Policy Editor
- Who can sign a Policy document
- Who can sign a Policy Negotiation objects

Figure 4 shows the UML class diagram for the *Authorization Service*. The *SimpleAuthorizationService* class implements the *AuthService* interface, which mandates the implementation of the following public method.

checkAuth()	Determine of the originator is authorized to perform a specific action.

The checkAuth() method executes in the context of the caller's thread. It performs its intended function without the use of blocking I/O operations and as such it will not cause the caller's thread to block for an extended period of time.

The *SimpleAuthorizationService* consults the contents of a XML Trusted Authorities file to determine the trusted certificate authority for a remote administrative domain. **Error! Reference source not found.** of this document contains a sample Trusted Authorities file.

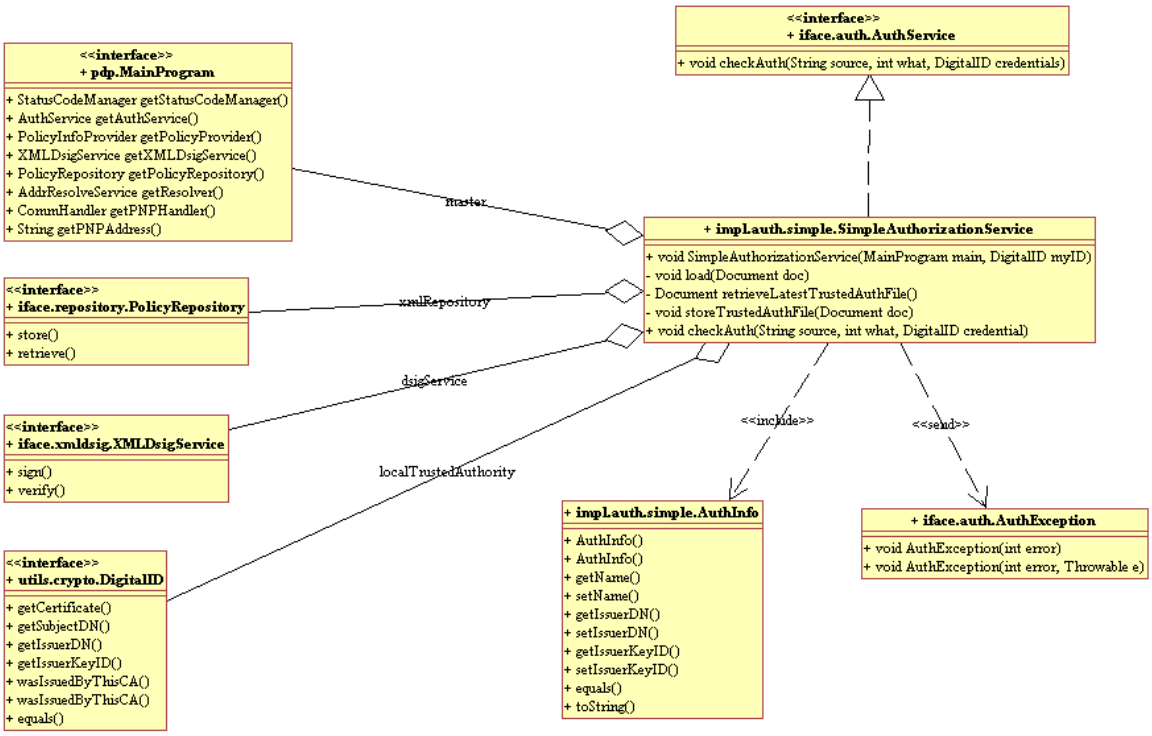

Figure 4 - Simple Authorization Service

5.4.1 Requirements

The *SimpleAuthorizationService* must fulfill the following requirements:

1. Retrieve the latest XML Trusted Authorities file from the policy repository at start up.

2. Verify the integrity, authenticity and authority of the XML Trusted Authorities file. When retrieved from the policy repository, the XML Trusted Authorities file must contain two nested digital signatures. The signature of the entity that created the file as the inner signature and the signature of the entity that validated the file as the outer signature.

3. Store the information contained in the XML Trusted Authorities file in an internal data structure.

4. Provide an authorization check service based on the administrative domain of the originator, the requested action, and the digital identifier of the originator.

In the future, the *SimpleAuthorizationService* should accept new versions of the XML Trusted Authorities file via the *Policy Submit Provider*.

5.4.2 Design

The *Main Program* creates a single instance of the *SimpleAuthorizationService* class. The lone *SimpleAuthorizationService* constructor takes two arguments: a reference to the *Main Program*, and a reference to a *DigitalID*, which contains the digital identifier of the local certificate authority. The *Main Program* provides the needed references to the *Policy Repository* and the *XML Digital Signature Service*. The *SimpleAuthorizationService* constructor retrieves the contents of the XML Trusted Authorities file from the policy repository; verifies the integrity and authenticity of the file; ensures that the file was created and validated by authorized entities; and loads the contents of the file into an internal data structure.

The *SimpleAuthorizationService* is ready to perform authorization checks once the constructor creates the internal trusted authorities data structure. The public checkAuth() method bases its decision on the expected administrative domain of the originator, the requested action as well as the digital identity of the originator. The expected administrative domain is expressed as an administrative domain name or null if the originator is expected to be from the local administrative domain. The *SimpleAuthorizationService* performs regular expression matches on the common name portion of the subject name encoded within the digital identifier (certificate).

5.4.2.1 Synchronization

The *SimpleAuthorizationService* class uses class level synchronization to ensure that only a single instance of the class is active within the system.

The private load() method uses object level synchronization to prevent the concurrent loading of the trusted authorities data structure by more than one thread. This synchronization is not needed at this time since only the constructor currently calls the private load() method. In the future however, the *SimpleAuthorizationService* class could accept new versions of the XML Trusted Authorities file via the *Policy Info Provider*. For this reason, the checkAuth() method also synchronizes its access to the trusted authorities data structure.

5.4.2.2 Exceptions

The *SimpleAuthorizationService* class throws the *AuthException* if it encounters the following difficulties:

- An attempt to instantiate more than one instance of the class.
- It cannot verify the digital signatures on the XML Trusted Authorities file.
- The XML Trusted Authorities file is not signed by an authorized entity.
- The XML Trusted Authorities file contains a structural, syntax or format error.
- The identified originator is not authorized to perform the requested action.

5.5 XML Address Resolution Service

The *XML Address Resolution Service* translates between administrative domain names and network layer information.

Figure 5 shows the UML class diagram for the *XML Address Resolution Service*. The *XMLAddressResolver* class implements the *AddrResolveService* interface, which mandates the implementation of the following public methods:

lookupAddress()	Translate an administrative domain name to network layer information.
reverseLookup()	Translate network layer information to an administrative domain name.

The lookupAddress() and reverseLookup() methods execute in the context of the caller's thread. They perform their intended functions without the use of blocking I/O operations and as such they will not cause the caller's thread to block for an extended period of time.

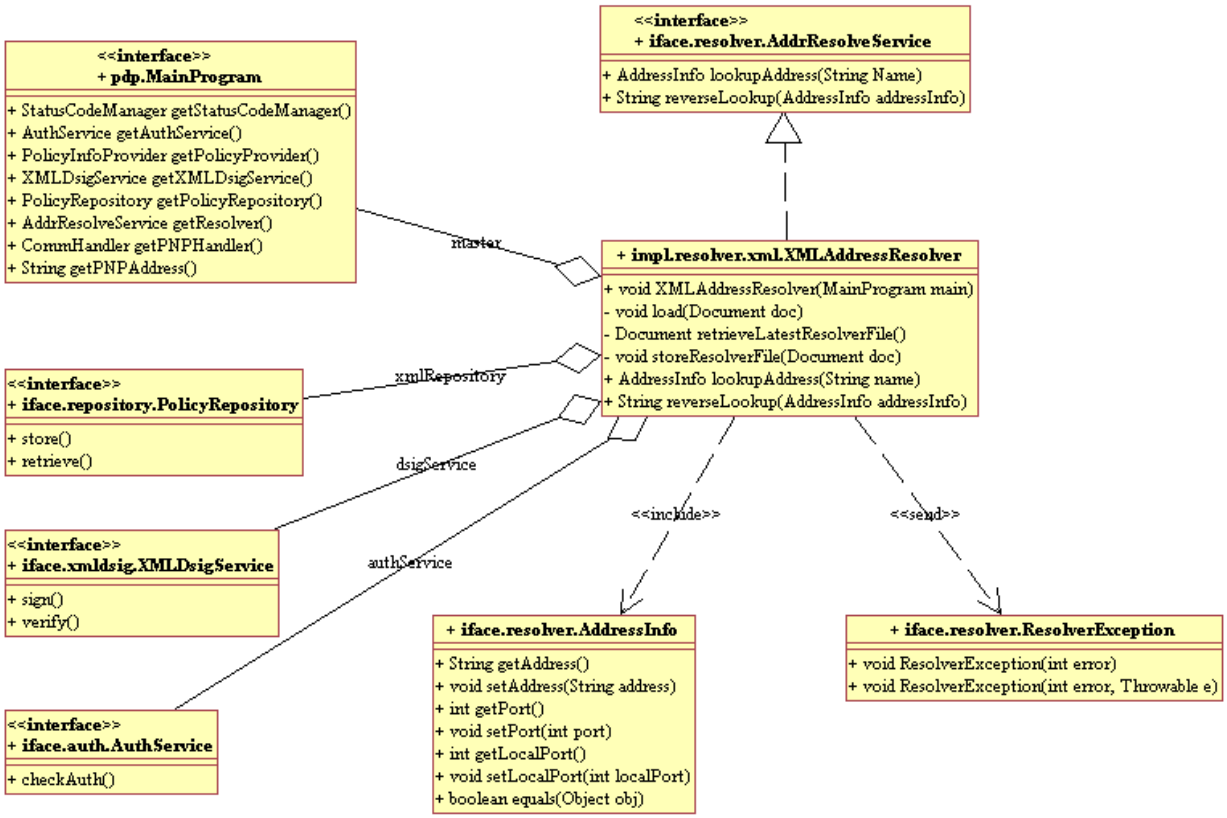

Figure 5 - XML Address Resolution Service UML Diagram

PBNM policies include references to remote administrative domains, but they do not explicitly identify how to communicate with these remote administrative domains. The *XMLAddressResolver* maintains an XML encoded Address Resolution Mapping file that describes how the local Policy Negotiation Proxy (PNP) communicates with the remote PNP for

the associated administrative domain. The network layer information includes the Internet Protocol (IP) address of the remote PNP, the transport layer port number of the local PNP, and the transport layer port number of the remote PNP. **Error! Reference source not found.** of this document contains a sample XML Address Resolution Mapping file.

5.5.1 Requirements

The *XMLAddressResolver* must fulfill the following requirements:

1. Retrieve the latest XML Address Resolution Mapping file from the policy repository at start up.

2. Verify the integrity, authenticity and authority of the XML Address Resolution Mapping file. When retrieved from the policy repository, the XML Address Resolution Mapping file must contain two nested digital signatures. The signature of the entity that created the file as the inner signature and the signature of the entity that validated the file as the outer signature.

3. Store the mapping information contained in the XML Address Resolution Mapping file in internal data structures.

4. Provide a forward translation service that maps an administrative domain name to network layer information.

5. Provide a reverse translation service that maps network layer information to an administrative domain name.

In the future, the *XMLAddressResolver* should accept new versions of the XML Address Resolution Mapping file via the *Policy Submit Provider*.

5.5.2 Design

The *Main Program* creates a single instance of the *XMLAddressResolver* class. The lone *XMLAddressResolver* constructor takes a reference to the *Main Program* as its argument. The *Main Program* provides the needed references to the *Policy Repository*, *XML Digital Signature Service* and the *Authorization Service*. The *XMLAddressResolver* constructor retrieves the contents of the XML Address Resolution Mapping file from the policy repository; verifies the integrity and authenticity of the file; ensures that the file was created and validated by authorized entities; and loads the contents of the file into a pair of internal data structures: one for forward lookups and one for reverse lookups.

The *XMLAddressResolver* is ready to perform address resolution operations once the constructor creates the necessary internal data structures. The public lookupAddress() method takes an administrative domain name as an argument and returns the associated network address information. The public reverseLookup() method takes network address information as an argument and returns the associated administrative domain name. Both methods return null if the lookup fails.

5.5.2.1 Synchronization

The *XMLAddressResolver* class uses class level synchronization to ensure that only a single instance of the class is active within the system.

The private load() method uses object level synchronization to prevent the concurrent loading of internal data structures by more than one thread. This synchronization is not needed at this time since only the constructor currently calls the private load() method. In the future however, the *XMLAddressResolver* class could accept new versions of the XML Address Resolution Mapping file via the *Policy Info Provider*.

Both the lookupAddress() and reverseLookup() methods are fully synchronized to prevent concurrent access to the internal data structures by more than one thread.

5.5.2.2 Exceptions

The *XMLAddressResolver* class throws the *ResolverException* if it encounters the following difficulties:

- An attempt to instantiate more than one instance of the class.
- It cannot verify the digital signatures on the XML Address Resolution Mapping file.
- The XML Address Resolution Mapping file is not signed by an authorized entity.
- The XML Address Resolution Mapping file contained a structural, syntax or format error.

5.6 Policy Info Provider

Authorized individuals within the local administrative domains may create or alter policies using the Policy Editor. Once compiled, the Policy Editor applies the security officer's digital signature to the policy document and submits the policy document to the PDP via a Secure Socket Layer (SSL) secured channel. The *Policy Info Provider* accepts new policy documents from the Policy Editor and routes them to registered *Policy Info Manager* objects.

Figure 6 shows the UML class diagram for the *Policy Info Provider*. The *PolicySubmitProvider* class implements the *PolicyInfoProvider* interface, which mandates the implementation of the following public methods:

register()	Register a Policy Information Manager with a specific type of policy document.
unregister()	Unregister a previously registered Policy Information Manager.

The register() and unregister() methods execute in the context of the caller's thread. They perform their intended functions without the use of blocking I/O operations and as such they will not cause the caller's thread to block for an extended period of time.

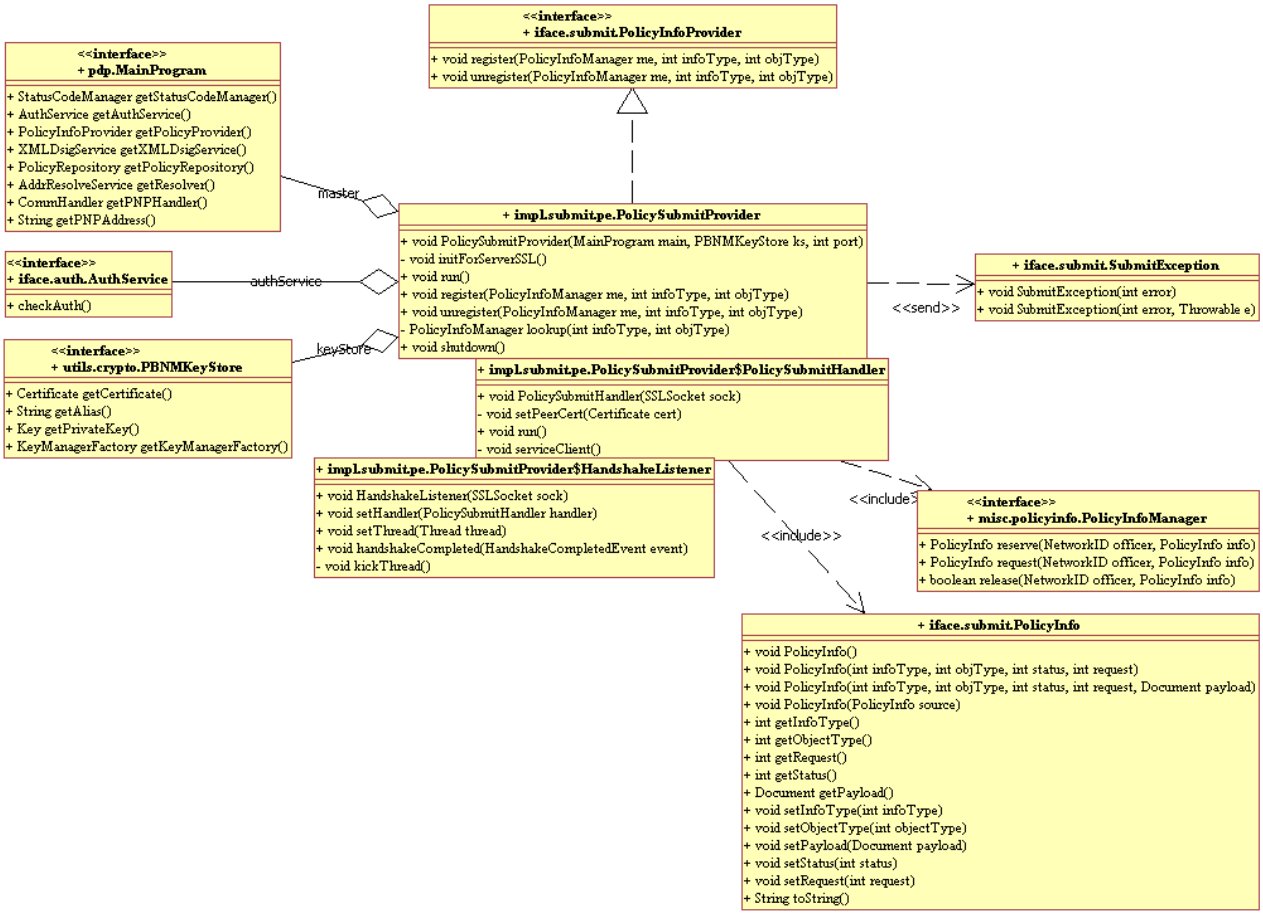

Figure 6 - Policy Info Provider UML Diagram

5.6.1 Requirements

The *PolicySubmitProvider* must meet the following requirements:

1. Accept registration requests from *Policy Info Manager* objects for specific policy documents.

2. Listen for SSL sessions from the Policy Editor.

3. Accept SSL sessions from the Policy Editor and service many concurrent instances of the Policy Editor.

4. Interact with interested *Policy Info Manager* objects on behalf of the Policy Editor.

5.6.2 Design

The *Main Program* creates a single instance of the *PolicySubmitProvider* class. The *PolicySubmitProvider* class runs in its own thread and therefore it implements the java.lang.Runnable interface.

The lone *PolicySubmitProvider* constructor takes three arguments: a reference to the *Main Program*, a reference to the *PBNMKeyStore* containing the digital credentials for the local *Policy*

Info Provider, and the transport port number to listen on for incoming SSL sessions. The *PolicySubmitProvider* retrieves a reference to the *Authorization Service* from the *Main Program*.

The *PolicySubmitProvider* accepts registration requests for specific types of policy documents and stores this registration information in an internal data structure. The registration information identifies the *Policy Info Manager* interested in receiving the specific type of policy document. Currently, only *PPU* objects register as *Policy Info Managers* with the *PolicySubmitProvider*.

The *PolicySubmitProvider* executes in its own thread where it waits for incoming SSL sessions. The *PolicySubmitProvider* creates a new instance of an inner class called the *PolicySubmitHandler* to service each individual SSL session. As well, the *PolicySubmitProvider* creates a new instance of an inner class called the *HandshakeListener* for each individual SSL session. Each *PolicySubmitHandler* executes in its own thread since many instances of the *PolicySubmitHandler* can execute concurrently. The *HandshakeListener* waits for the SSL handshake to complete before attempting to obtain the certificate used by the Policy Editor to authenticate the SSL session. When the certificate is available, the *HandshakeListener* starts the *PolicySubmitHandler* thread.

5.6.2.1 Handshake Listener

The *HandshakeListener* class implements the javax.net.ssl.HandshakeCompletedListener interface. This interface requires that the Policy Submit Provider implement the following public method:

handshakeCompleted()	An indication that the SSL handshake completed.

The *PolicySubmitProvider* provides a reference to the *PolicySubmitHandler* object and the associated thread by invoking the setHandler() and setThread() methods on the *HandshakeListener* object. When the SSL handshake completes, the *HandshakeListener* obtains the certificate used by the Policy Editor[3] to authenticate the SSL session. Once the *HandshakeListener* obtains all three pieces of information, it provides the certificate to the *PolicySubmitHandler* by invoking the *PolicySubmitHandler* setPeerCert() method and calls the start() method of the supplied thread to start the corresponding *PolicySubmitHandler* thread.

5.6.2.2 Policy Submit Handler

The *PolicySubmitHandler* runs in its own thread and therefore it implements the java.lang.Runnable interface.

The *PolicySubmitHandler* requests an authorization check from the *AuthorizationService* to verify that the SSL session was initiated by an authorized individual. The *PolicySubmitHandler* retrieves the reference for the registered *Policy Info Manager* from its internal data structure and relays *PolicyInfo* objects between the Policy Editor and the *Policy Info Manager*. The *PolicySubmitHandler* does not interpret the payload contained within the *PolicyInfo* objects however. The *PolicySubmitHandler* continues to relay *PolicyInfo* objects until a response from the *Policy Info Manager* indicates an erroneous condition or until the Policy Editor closes the SSL session.

5.6.2.3 Synchronization

The *PolicySubmitProvider* uses class level synchronization to ensure that only a single instance of the class is active within the system.

[3] The Policy Editor makes use of the certificate owned by the authorized individual.

The *HandshakeListener* synchronizes the start of the *PolicySubmitHandler* thread with the availability of the certificate used to authenticate the SSL session from the Policy Editor. The *HandshakeListener* object also uses object level synchronization to ensure that it only attempts to start the *PolicySubmitHandler* thread once.

The register(), unregister() and lookup() methods are fully synchronized to prevent concurrent access to the list of registered *Policy Info Managers* by more than one thread.

5.6.2.4 Exceptions

The *PolicySubmitProvider* class throws the *SubmitException* if it encounters the following difficulties:

- An attempt to instantiate more than one instance of the class.
- A failure to initialize the SSL server context.
- An attempt by a *Policy Info Manager* to register for a type of policy document already claimed by another *Policy Info Manager* object.

5.6.3 Future Considerations

Currently the Policy Editor only supports Inter-Domain Security Policies. In the future, the Policy Editor should also support different types of policies as well as the editing of the XML Address Resolution Mapping file used by the XML Address Resolution Service and the XML Trusted Authorities file used by the Simple Authorization Service. The generic design of the *PolicySubmitProvider* will support these different types of policy documents without any modifications.

Currently the Policy Editor stores policy documents on its local disk and submits new or modified policy documents to the PDP. In the future, policy documents should only reside in the Policy Repository and the Policy Editor should acquire the latest version of a policy document from the PDP prior to editing. The generic design of the *PolicySubmitProvider* will support the retrieval of the policy document without any modifications.

5.7 Comm Handler

The *Comm Handler* manages the communication between the PDP and the PNP.

Figure 7 shows the UML class diagram for the *Comm Handler*. The *PNPHandler* class implements the *CommHandler* interface, which mandates the implementation of the following public methods:

submit()	Submit a PNU to the *CommHandler* from the PPU.
isConnected()	Determine if the PDP is connected to the PNP.
close()	Close the communication channel to the PNP.

These methods execute in the context of the caller's thread – the *PPU*. They perform their intended function without the use of blocking I/O operations and as such it will not cause the caller's thread to block for an extended period of time.

The *PNPHandler* class also implements the *AlertListener* interface, which mandates the implementation of the following public method:

alert()	Submit a PNU to the *CommHandler* from the PNP.

The alert() method executes in the context of the caller's thread. It performs its intended function without the use of blocking I/O operations and as such it will not cause the caller's thread to block for an extended period of time. However, the alert() method does call the alert() method of the associated *PPU*, which must also not cause the caller's thread to block on I/O operations.

The *PNPHandler* instantiates a distinct *CommWorker* object to establish and maintain the communication channel with the PNP. The *CommWorker* interface requires that the implementing class provide the following public methods:

isConnected()	Determine if the PDP is connected to the PNP.
submit()	Submit a PNU to the *CommWorker* from the PPU.
close()	Close the communication channel to the PNP.

These methods execute in the context of the caller's thread. They perform their intended functions without the use of blocking I/O operations and as such they will not cause the caller's thread to block for an extended period of time.

The PDP currently includes a single implementation of the *CommWorker* based on the JGroups communication framework [JGROUPS]. In the future, different *CommWorker* implementations (i.e. Transport Layer Security, Session Initiation Protocol) can be added to the system.

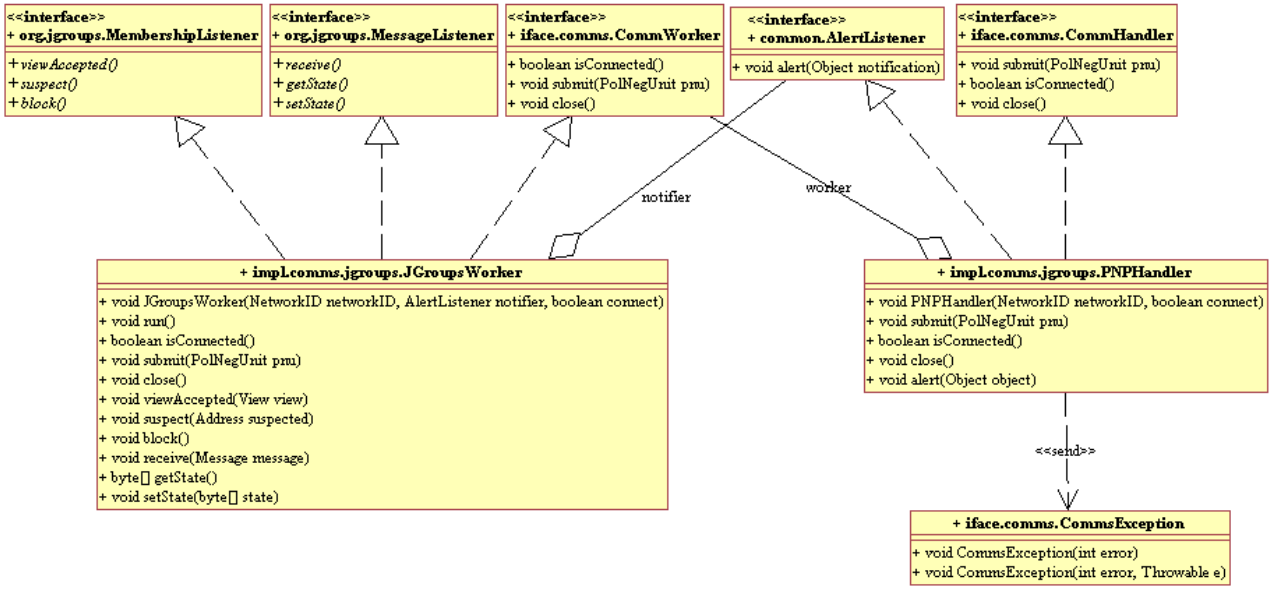

Figure 7 - Comm Handler UML Diagram

5.7.1 Requirements

The *PNPHandler* must fulfill the following requirements:

1. Process requests from Policy Processing Units (PPU) to engage in communication with a remote administrative domain.

2. Multiplex PNUs for numerous remote administrative domains within a single PDP/PNP communication channel.

3. Process requests from PPUs to disengage from a remote administrative domain.

4. Forward PNUs received from the PPU to the PNP.

5. Forward PNUs received from the PNP to the appropriate PPU.

5.7.2 Design

The *Main Program* creates a single instance of the *PNPHandler* class. The lone *PNPHandler* constructor requires the network identity of the PNP and a boolean flag as its arguments. The boolean flag instructs the *PNPHandler* to actively connect to the PNP. The *PNPHandler* creates a single instance of a *CommWorker*, which in the current system is a *JGroupsWorker*. The *CommWorker* interface extends the java.lang.Runnable interface, which causes an implementing class to run in its own thread.

The *PNPHandler* processes control PNUs from *PPU* objects to engage and disengage from a remote administrative domain. The control PNU specifies the type of policy to be negotiated, identifies the *PPU* associated with the policy, provides the network information required by the PNP to communicate with the remote administrative domain, and assigns a connection identifier for use as a reference within subsequent data PNUs. The *PNPHandler* uses this information to create an internal routing table. The *PNPHandler* forwards slightly modified versions of control PNUs to the PNP through the *CommWorker*. The *PNPHandler* relays data PNUs between the PNP and the responsible *PPU* using its internal routing table.

The *JGroupsWorker* establishes and maintains the communication channel between the PDP and the PNP. The *JGroupsWorker* simply relays PNUs between the *PNPHandler* and the PNP. It accepts outgoing PNUs from the *PNPHandler* and delivers them to the PNP, and it accepts incoming PNUs from the PNP and delivers them to the *PNPHandler*.

5.7.2.1 Synchronization

The *JGroupsWorker* class synchronizes access to a shared queue where outgoing PNUs are deposited by the *PNPHandler* thread and retrieved by the *JGroupsWorker* thread.

5.7.2.2 Exceptions

The *PNPHandler* class throws the *CommsException* when a *PPU* submits an unknown type of object – not a valid PNU.

5.8 Policy Server

The *Policy Server* collects low level policies from *Policy Processing Units* (PPUs) and disseminates the low level policies to Policy Enforcement Point (PEP) devices through *Policy Agents*. The PBNM system currently supports a single *Policy Agent* based on COPS-PR implementation from the University of Murcia called UMU-JCOPS.

Figure 7 shows the UML class diagram for the *Policy Server* as implemented by the *PolicyServer* class. The *PolicyServer* class makes use of the Singleton pattern to ensure that only a single instance of the class executes within the PBNM system. The *PolicyServer* class provides public methods for initialization, to register device roles, as well as to submit policy updates.

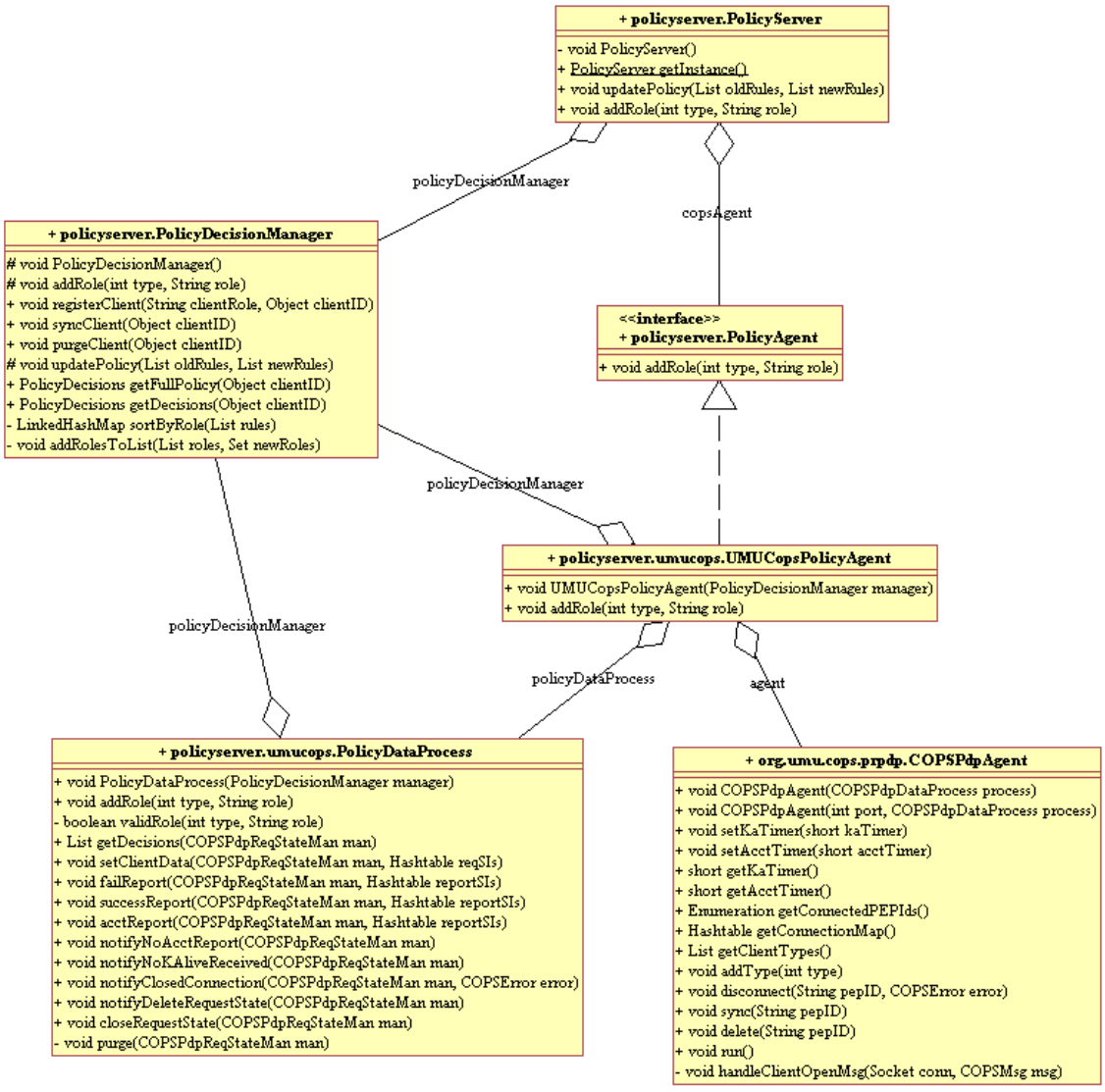

Figure 8 – Policy Server UML Diagram

5.8.1 Requirements

The *PolicyServer* must fulfill the following requirements:

1. Create the *PolicyAgent* objects as required – i.e. UMU-JCOPS Agent.
2. Accept requests from PPUs to register device roles.
3. Accept policy updates from PPUs.
4. Disseminate policy updates to *PolicyAgents*.

5.8.2 Design

The *PolicyServer* class creates *PolicyAgent* objects for each supported policy dissemination method. The *PolicyServer* accepts requests from PPU objects within the PBNM system to register devices roles that the *PolicyServer* must support. The *PolicyServer* also receives policy updates from PPU objects as a list of old policies and new policies.

The *PolicyServer* relies on a protected class called the *PolicyDecisionManager* to determine the differences between the old policies and new policies and to formulate a list of policy decisions. The *PolicyDecisionManager* keeps track of the roles supported by the *PolicyServer* and accepts requests from *PolicyAgent* objects to register PEP clients for specific roles. The *PolicyDecisionManager* manages full policies for a specific role and maintains policy decisions for individual PEP clients based on their identified roles. Once a PEP client acquires a complete policy document, the *PolicyDecisionManager* simply provides deltas in the form of add and remove decisions.

The *PolicyServer* relies on *PolicyAgent* objects to disseminate policy decisions to network PEP devices. Currently network devices must acquire their device level policies through the COPS-PR protocol. The *UMUCopsPolicyAgent* class implements the *PolicyAgent* interface, which mandates the implementation of the following public methods:

addRole()	Add a new device role. This informs the *PolicyAgent* that PEP devices will request policies for that specific role.

The *UMUCopsPolicyAgent* interacts with the UMU-JCOPS implementation through a single class called the *COPSPdpAgent*. The UMU-JCOPS also requires the extension of an abstract class called the *COPSPdpDataProcess*, which defines the call-back methods that its sub-class must implement in order to interact properly with the UMU-JCOPS protocol stack. The *PolicyDataProcess* class extends the *COPSPdpDataProcess* class to provide the needed functionality.

The *COPSPdpAgent* object listens for connections from COPS clients, processes COPS Client-Open messages to confirm support for the COPS Client-Type, and creates the necessary COPS objects needed to service the COPS client. Further interactions between the UMU-JCOPS protocol stack and the *PolicyServer* are facilitated through call-back method calls on the *PolicyDataProcess* object. The *PolicyDataProcess* object possesses a reference to the *PolicyDecisionManager* object and obtains complete policy documents as well as policy decisions directly from the *PolicyDecisionManager* object.

5.8.2.1 Low-Level Policy Objects

The PBNM system provides an interface called *PolicyManager* to manage low-level policies. The abstract class *DOMPolicyManager* implements the *PolicyManager* interface and provides a DOM specific *PolicyManager*. The system also provides concrete *PolicyManager* classes for

different types of low-level policies such as IPSec, Firewall, Routing and DNS. The *DOMPolicyManager* provides static methods for creating policy elements such as IP Addresses, Selectors and Time Periods.

The *PolicyDecisionManager* creates concrete *PolicyManager* objects as required to manage policies for specific roles. The *PolicyManager* accepts policy updates, determines the difference between the old policies and the new policies and produces deltas in the form of a list of policy (add/remove) decisions.

Figure 9 shows the UML class diagram for the *PolicyManager* interface, *DOMPolicyManager* abstract class as well as for the *FWPolicyManager* concrete class. Figure 9 also illustrates the *PolicyDecisions* class that the *PolicyManager* class uses to return its policy decisions.

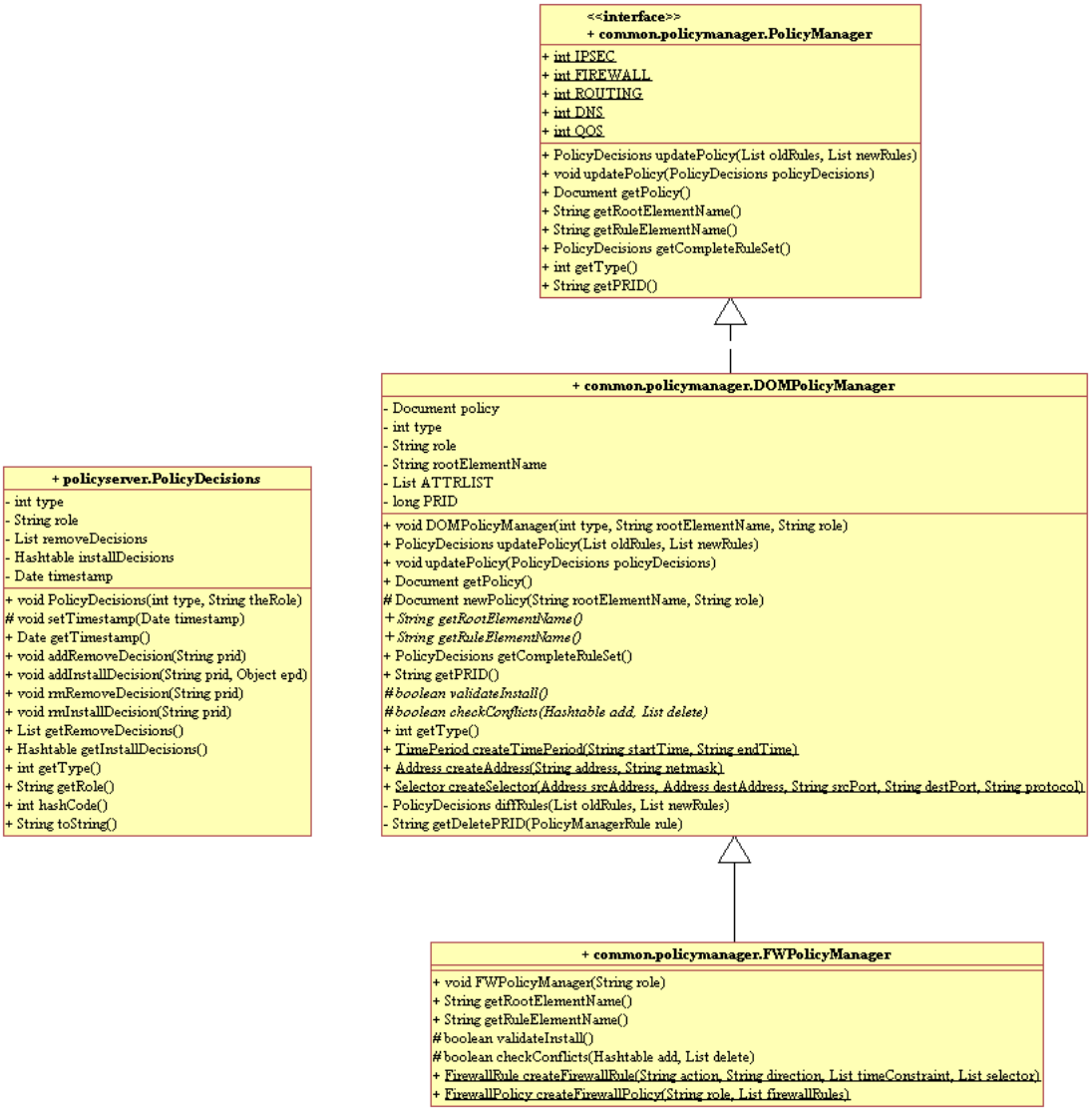

Figure 9 - Policy Manager UML Diagram

5.8.2.2 Low-Level Policy Elements

The PBNM system provides a series of classes to create and maintain low-level policy elements such as rules, IP Addresses, Selectors and Time Periods. Each policy element class must

implement a method called *toDOM()* to convert the object to a DOM object and a method called *getXpathConditions()* to produce a list of XPATH query strings which uniquely identify the object. Policy elements based on rules must also implement a method called *compare()* to compare itself against another rule and a method called *getXpathQuery()* to produce an XPATH query string which uniquely identifies the object.

Figure 10 shows the UML class diagram for the low-level policy element classes.

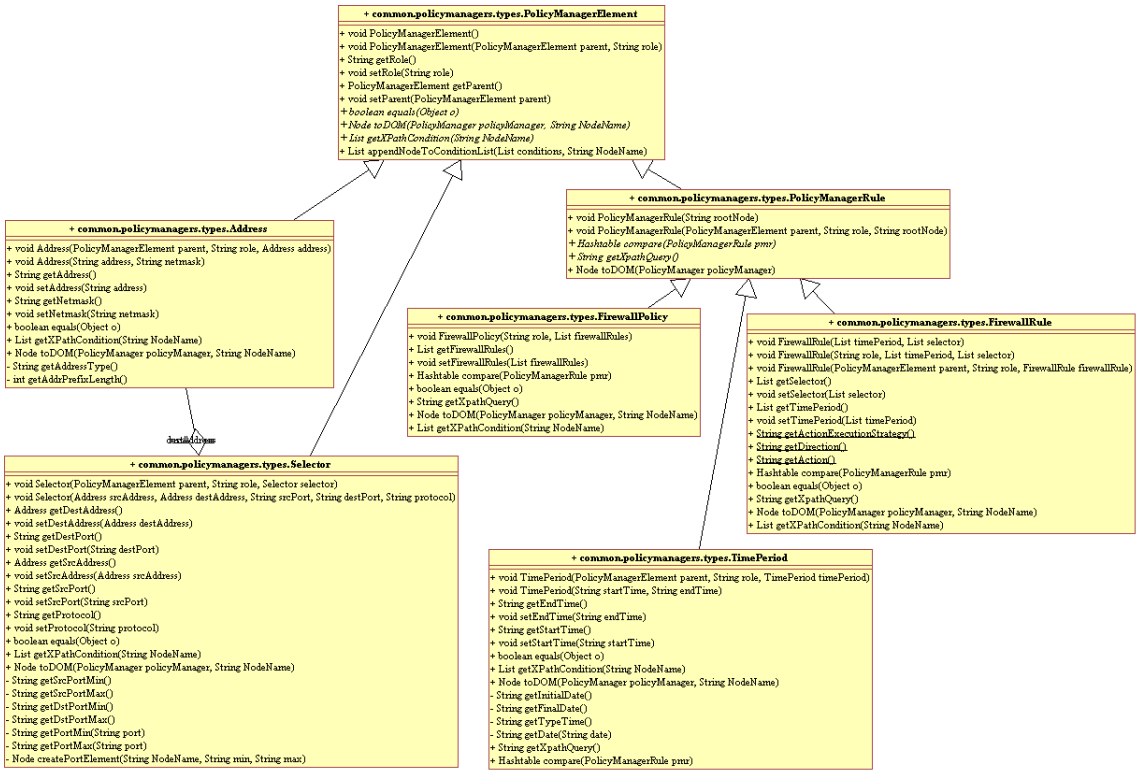

Figure 10 - Policy Element UML Diagram

5.8.2.3 The COPSPdpDataProcess Abstract Class

The *PolicyServer* implements a *COPSPdpDataProcess* sub-class to manage and maintain low-level policies. The *COPSPdpDataProcess* abstract class mandates the implementation of the following methods:

addRole()	Add a new device role. PEP devices will request policies for this specific role.
getDecisions()	Retrieve outstanding policy decisions (remove, install).
setClientData()	Set the COPS client data contained in the Client Specific Information (Client-SI). This identifies the device role and indicates whether the COPS client is requesting a complete state update. UMU-JCOPS implements a subset of RFC-3318

	(Framework Policy Information Base) within the Client-SI data.
failReport()	Process a COPS Failure report.
successReport()	Process a COPS Success report.
acctReport()	Process a COPS Accounting report.
notifyNoAcctReport()	An expected accounting report was not received when expected.
notifyNoKAliveReceived()	An expected Keepalive message was not received.
notifyClosedConnection()	The COPS client closed the connection.
notifyDeleteRequestState()	The COPS client deleted its request state.

5.8.2.4 The PolicyDataProcess Class

The *PolicyDataProcess* class maintains mappings between COPS clients and their declared roles and only supplies decisions associated with these declared roles. *PolicyDataProcess* relies exclusively on the *PolicyDecisionManager* to manage policy decisions. It registers COPS clients with the *PolicyDecisionManager* when COPS clients connect and identify their roles.

5.8.2.5 Synchronization

The *PolicyDataProcess* object executes in various threads that include PPUs threads as well as threads created by UMU-JCOPS objects. The *PolicyDataProcess* object makes use of object based synchronization to prevent concurrent access to its internal data structures.

5.8.2.6 Exceptions

The *PolicyServer* class throws the *PolicyServerException* if it exhibits difficulties in instantiating *PolicyAgent* objects or submitting policy decisions to *PolicyAgent* objects.

5.9 Policy Processing Unit

The Policy Processing Unit (PPU) forms the core of the PDP and is the only component that possesses knowledge of policy structure, syntax and encoding. The PDP requires the implementation of a distinct *PPU* class for each type of policy supported by the system.

Figure 11 illustrates how a distinct *PPU* that implements negotiable inter-domain security policies is implemented by extending a pair of abstract Java classes.

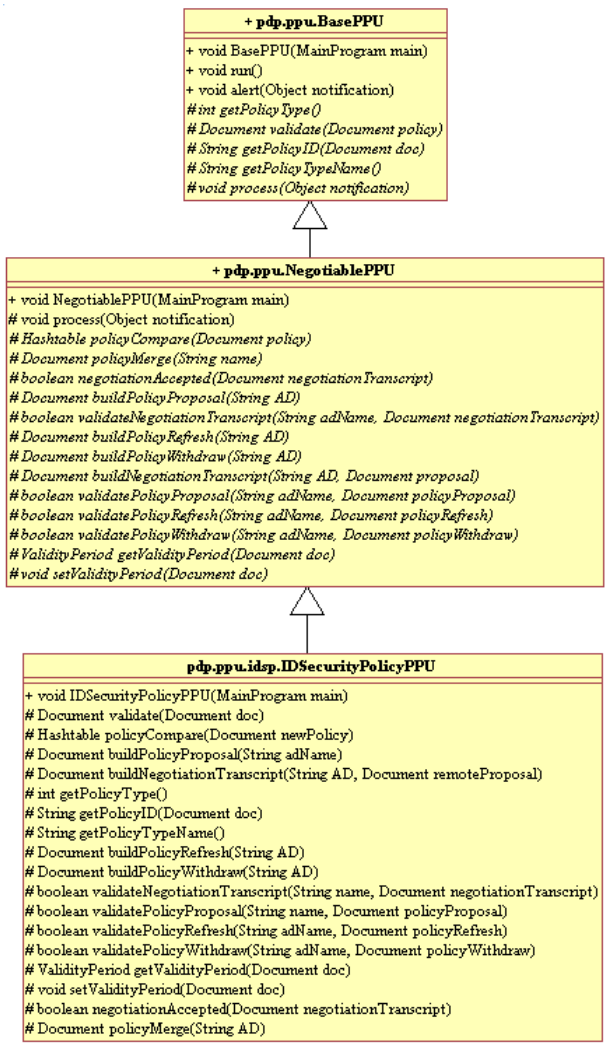

Figure 11 - Creating a Distinct PPU

The *BasePPU* abstract class provides the generic capabilities required by all *PPU* implementations and as such it must be included as a super class by all *PPU* implementations. The *BasePPU* abstract class accepts new policies from the Policy Editor via the *Policy Info Provider*. The *BasePPU* abstract class does not perform any policy specific processing. As such, the *BasePPU* abstract class requires that one of its sub classes implement the following methods:

getPolicyType()	Return the type of policy that this PPU processes.
getPolicyTypeName()	Return the name of the type of policy that this PPU processes.
validate()	Validate a policy document.
getPolicyID()	Return the unique identifier from a policy document.
process()	Process an asynchronous event.

The *NegotiablePPU* abstract class extends the *BasePPU.* Since it negotiates policies with foreign administrative domains, it must be included as a super class by all *PPU* implementations that require inter-domain policy negotiation. The *NegotiablePPU* class implements the policy negotiation protocol described in the PBNM System Design Document [PBNM]. The *NegotiablePPU* abstract class does not perform any policy specific processing. As such, the *NegotiablePPU* abstract class requires that one of its sub classes implement the following methods:

policyCompare()	Compare the existing policy with the newly received policy.
buildPolicyProposal()	Build a Policy Proposal object.
buildPolicyRefresh()	Build a Policy Refresh object.
buildPolicyWithdraw()	Build a Policy Withdraw object.
buildNegotiationTranscript()	Build a Negotiation Transcript object.
validatePolicyProposal()	Validate a Policy Proposal object.
validateNegotiationTranscript()	Validate a Negotiation Transcript object.
validatePolicyRefresh()	Validate a Policy Refresh object.
validatePolicyWithdraw()	Validate a Policy Withdraw object.
getValidityPeriod()	Return the validity period for a policy object.
setValidityPeriod()	Set the validity period in a policy object.
negotiationAccepted()	Look inside the Negotiation Transcript objects to determine if the negotiation was successful.
policyMerge()	Create a Merged Policy object from two Negotiation Transcript objects.

The *IDSecurityPolicyPPU* class extends the *NegotiablePPU* to implement a complete *PPU* that negotiates inter-domain security policies with foreign administrative domains. The *IDSecurityPolicyPPU* class simply provides the policy specific methods mandated by its super classes (*BasePPU* and *NegotiablePPU*) and as such is the only class within the PDP that possesses any knowledge of the structure, format and encoding of inter-domain security policies.

The PDP can be easily extended to process a different type of negotiated policy by implementing a distinct *PPU* class that extends the *NegotiablePPU* class. To support a static (non-negotiated) policy, the PDP must implement a *StaticPPU* abstract class that extends the *BasePPU* class in addition to a distinct *PPU* class that extends the *StaticPPU* abstract class.

5.9.1 PPU Design Overview

Figure 12 shows the UML class diagram for a complete *Negotiable PPU* implemented by the *IDSecurityPolicyPPU* class. The *BasePPU* class acquires references to several infrastructure

services from the *Main Program*. Similarly, the *NegotiablePPU* class acquires references to the *Comm Handler* and the *XML Address Resolution Service* from the *Main Program*. The *BasePPU* class includes an inner class called the *PolicyConsumer*, and the *NegotiablePPU* includes an inner class called the *Negotiator*. Both the *BasePPU* class and the *Negotiator* class implement the java.lang.Runnable interface, which provides a different thread of execution for each object, and the *AlertListener* interface, which facilitates the delivery of asynchronous events from objects executing in a different thread. The *AlertListener* interface provides the inter-thread communication mechanism between the *Negotiable PPU* executing in the *BasePPU* thread and each *Negotiator* executing in its respective thread.

The *PolicyConsumer* implements the *PolicyInfoConsumer* interface in order to receive policy documents from the Policy Editor via the *Policy Info Provider*. The *PolicyConsumer* makes use of a *PolicyInfoSessionManager* to manage the session between the *PPU* and the *Policy Info Provider*.

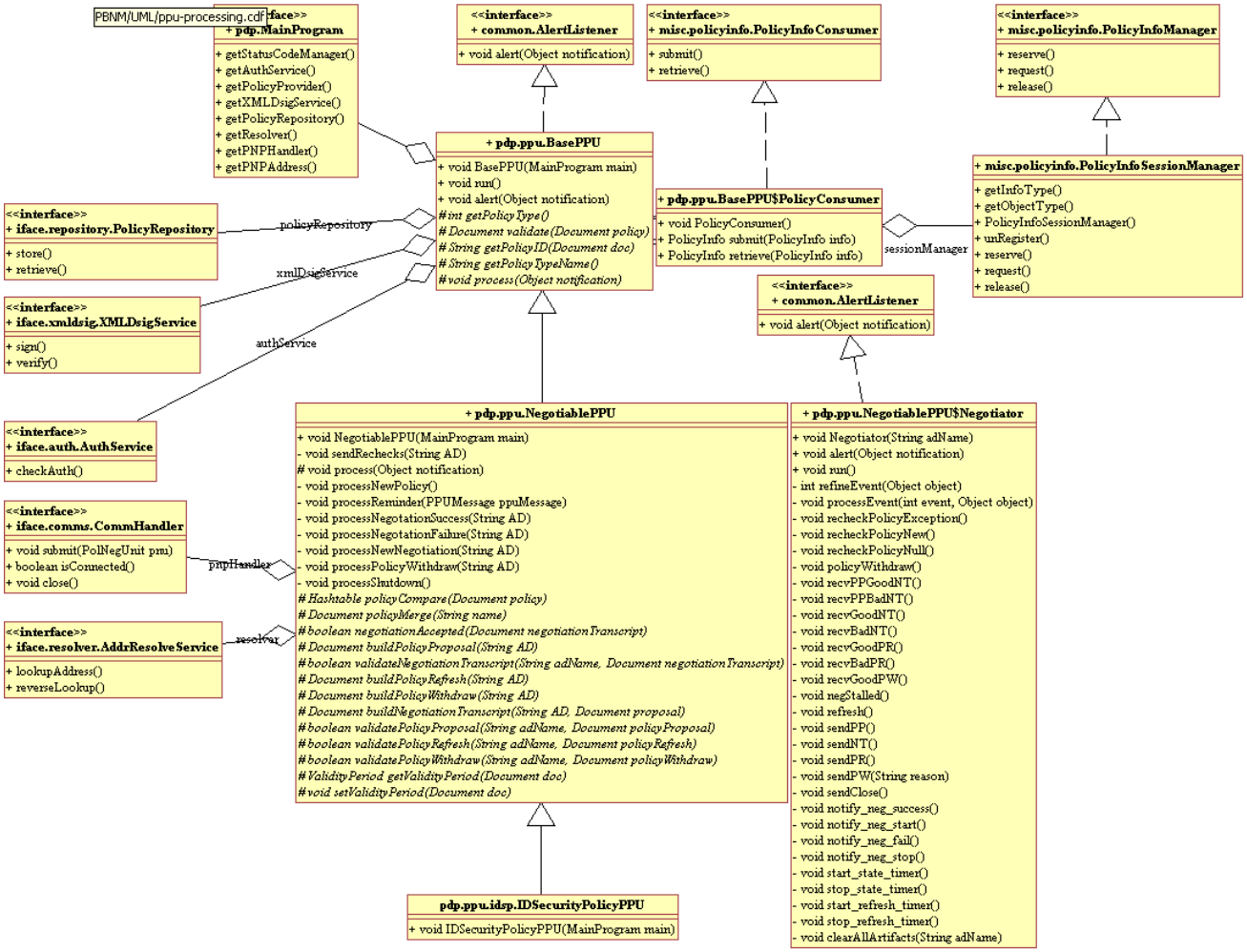

Figure 12 - Negotiable PPU UML Diagram

5.9.2 The Base PPU

The *BasePPU* provides the generic capabilities required by all PPU implementations.

5.9.2.1 Requirements

The *BasePPU* must meet the following requirements:

1. Accept and validate new policy documents from the Policy Editor via the *Policy Info Provider*.

2. Dispatch asynchronous events to subclasses for processing.

In the future, the *BasePPU* should also accept commands from the Policy Console via the *Main Program*.

5.9.2.2 Design

The *BasePPU* class implements the *AlertListener* interface, which mandates the implementation of the following public method:

alert()	Process the alert associated with the supplied object.

The alert() method executes in the context of the caller's thread – a *Negotiator*, a *Reminder* or the *PolicyConsumer*. It performs its intended function without the use of blocking I/O operations and as such it will not cause the caller's thread to block for an extended period of time.

Since the *BasePPU* class is abstract, it is only instantiated when the *Main Program* creates a distinct and concrete *PPU* object. The *BasePPU* class implements the java.lang.Runnable interface and therefore the resulting distinct PPU object runs in its own thread.

The *BasePPU* maintains the current and new policy document within *LockableDocument* objects.

The lone *BasePPU* constructor takes a single argument: a reference to the *Main Program*. The *BasePPU* retrieves a reference to the *Policy Repository*, the *XML Digital Signature Service*, the *Authorization Service*, and the *Policy Info Provider* from the *Main Program*. The *BasePPU* determines the type of policy that the distinct *PPU* object implements. Finally, the *BasePPU* object creates an inner class called the *PolicyConsumer* to process policy documents submitted by the Policy Editor.

The *BasePPU* provides the *BasePPU* execution thread for the distinct *PPU* object. Within its *run()* method, the *BasePPU* continually retrieves objects from a *SynchronizedQueue* and calls upon the abstract *process()* method to process the asynchronous event contained within the object. The *BasePPU* does not interpret the asynchronous events contained within these objects.

Policy Consumer

The *PolicyConsumer* processes policy documents submitted by the Policy Editor via the *Policy Info Provider*. The *PolicyConsumer* makes use of a *PolicyInfoSessionManager* to manage the session with the *Policy Info Provider*. The *PolicyInfoSessionManager* registers with the *Policy Info Provider* to receive the appropriate type of policy documents from the Policy Editor.

The *PolicyConsumer* class implements the *PolicyInfoConsumer* interface, which mandates the implementation of the following public methods:

retrieve()	Retrieve and supply the latest policy document from the *Policy Repository*. This method is not currently implemented.

submit()	Process a newly edited policy document from the Policy Editor.

The retrieve() and submit() methods execute in the context of the caller's thread –the *PolicyInfoSessionManager* on behalf of the *Policy Info Provider*. They perform their intended functions with the aid of the *Policy Repository* and as such they will cause the caller's thread to block until the I/O operation completes.

When the *PolicyConsumer* receives a policy document, it verifies the digital signature on the document, it ensures that the policy document was signed by an authorized individual and it validates[4] the policy document against the appropriate policy specification. If the policy document is authentic, authorized and valid, the *PolicyConsumer* applies the PDP's digital signature to the policy document and stores two copies in the *Policy Repository* – one copy based on its unique identifier and one copy as the latest policy document. The *PolicyConsumer* stores a reference to the new policy document in a *LockableDocument* and notifies the *BasePPU* via the *BasePPU* alert() method.

Synchronization

Access to the *BasePPU SynchronizedQueue* is automatically synchronized.

The *PolicyConsumer* applies a write lock to the *LockableDocument* that stores the new policy document within the *BasePPU*. The write lock ensures that the previously submitted policy document was fully processed before the new policy document is submitted to the *BasePPU*.

Exceptions

The *BasePPU* class throws the *PolicyProcessingException* if it encounters the following difficulties:

- The registration with the *Policy Info Provider* fails.

5.9.3 The Negotiable PPU

The *Negotiable PPU* class implements the policy negotiation protocol described in the PBNM System Design Document [PBNM].

5.9.3.1 Requirements

The *NegotiablePPU* class must meet the following requirements:

1. Process new policy notifications from the *PolicyConsumer*.
2. Negotiate policies with remote administrative domains.
3. Submit merged policy information to the *Policy Server* as required.
4. Retract merged policy information from the *Policy Server* as required.
5. Process shutdown notifications from the *Main Program*.

[4] The *PolicyConsumer* calls an abstract method implemented by the distinct PPU class to perform the validation.

5.9.3.2 Design

The *NegotiablePPU* class extends the *BasePPU* class and runs in the context of the *BasePPU* thread. The *NegotiablePPU* includes an inner class called the *Negotiator* that performs the policy negotiation. Each *Negotiator* executes within its own thread and interacts with a single remote administrative domain. The *NegotiablePPU,* in the context of the *BasePPU* thread, communicates with its numerous *Negotiator* objects within their respective threads using the *AlertListener* interface. This inter-thread communication is facilitated with a *SynchronizedQueue* – one *SynchronizedQueue* for each *PPU* thread.

The *NegotiablePPU* maintains several shared lists within *LockableHashtable* objects. They include:

- The priority associated with each remote administrative domain.
- The current negotiation state for each remote administrative domain.
- The last Policy Proposal sent to each remote administrative domain.
- The last Policy Proposal received by each remote administrative domain.
- The Negotiation Transcript sent to each remote administrative domain.
- The Negotiation Transcript received by each remote administrative domain.
- The Merged Policy created for each remote administrative domain.

When the *NegotiablePPU* receives a new policy document, it invokes the abstract *policyCompare()* to compare the new policy against the existing policy. This comparison yields a list of deleted administrative domains, a list of modified administrative domains, and a list of new administrative domains. For a deleted administrative domain, the *NegotiablePPU* sends a policy withdraw notification to the associated *Negotiator* and informs all lower priority administrative domains to recheck their policy. For modified administrative domain, the *NegotiablePPU* sends a recheck policy notification to the associated *Negotiator*. For a new administrative domain, the *NegotiablePPU* creates a new *Negotiator*, starts the corresponding *Negotiator* thread, and sends a recheck policy notification to the newly created *Negotiator*.

When the *NegotiablePPU* receives a notification from a *Negotiator* that a negotiation sequence completed successfully, the *NegotiablePPU* provides the merged policy information for the associated administrative domain to the *Policy Server*. The *NegotiablePPU* also sends a recheck policy notification to all *Negotiators* associated with a lower priority administrative domain.

When the *NegotiablePPU* receives a notification from a *Negotiator* that a negotiation sequence was restarted, the *NegotiablePPU* sets a time limit on the duration of a negotiation sequence. When the associated *Reminder* expires, the *NegotiablePPU* retracts the merged policy information, if any, for the associated administrative domain from the *Policy Server*.

When the *NegotiablePPU* receives a notification from a *Negotiator* class that a foreign administrative domain withdrew its previously negotiated policy, the *NegotiablePPU* retracts the merged policy information for the associated administrative domain from the *Policy Server*. The *NegotiablePPU* also sends a recheck policy notification to all the *Negotiators* associated with a lower priority administrative domain.

The Negotiator

The *Negotiator* class implements the policy negotiation protocol described by the policy negotiation state transition diagram that appears in the PBNM System Design Document [PBNM]. A detailed state transition table for the policy negotiation protocol appears in **Error! Reference source not found.** of this document.

Each instance of the *Negotiator* class runs in its own thread and undertakes a negotiation sequence with a distinct remote administrative domain. The *Negotiator,* executing within its own thread, and the *NegotiablePPU,* executing in the context of the *BasePPU* thread, communicate using the *AlertListener* interface. This inter-thread communication is facilitated with a *SynchronizedQueue* – one *SynchronizedQueue* for each *PPU* thread.

The *Negotiator* accepts all internal requests and external policy negotiation objects from its *SynchronizedQueue*. Internal requests include requests to recheck a policy, requests to withdraw a policy as well as *Reminder* notifications. External policy objects are the policy objects generated by a remote administrative domain and received as part of the policy negotiation sequence. The private refineEvent() method takes an object from the *SynchronizedQueue* and refines it to produce one of the events identified in the top row of the state transition table. The private processEvent() method takes a refined event and the associated object and dispatches it to the processing method associated with the refined event.

The *Negotiator* sends notifications to the *NegotiablePPU* to announce the following events:

- A negotiation sequence completed successfully
- A negotiation sequence restarted. This is usually caused by the reception of a Policy Proposal object from the remote administrative domain.
- A negotiation sequence completed in failure.
- The remote administrative domain withdrew its policy.

Synchronization

The *NegotiablePPU* and *Negotiator* classes synchronize access to shared lists by setting read and write locks as required on the associated *LockableHashtable* objects. The *NegotiablePPU* and *Negotiator* classes apply these read/write locks as required prior to invoking the abstract methods within the distinct *PPU* implementation. The distinct *PPU* implementation is not aware that these objects are lockable.

Access to the various *SynchronizedQueue* objects are automatically synchronized.

Exceptions

The *NegotaiblePPU* class throws the *PolicyProcessingException* if the registration with the *Policy Info Provider* fails.

The *Negotiator* class throws the *ResolverException* if it cannot acquire the network address information for a specific remote administration domain.

The *Negotiator* class throws the *UnknownHostException* if the network address supplied for the administrative domain is incorrect.

5.9.4 The Inter-Domain Security Policy PPU

The *Inter-Domain Security Policy PPU* implements the policy specification described in [IDSP].

5.9.4.1 Requirements

The *Inter-Domain Security Policy PPU* must meet the following requirements:

1. Validate inter-domain security policy documents.

2. Compare two inter-domain security policy documents and report on the differences
3. Produce inter-domain security policy negotiation objects.
4. Validate inter-domain security policy negotiation objects.
5. Produce merged inter-domain security policy objects.

5.9.4.2 Design

The *IDSecurityPolicyPPU* class extends the *NegotiablePPU* class. The resulting *PPU* object runs in the context of either the *BasePPU* thread or a *Negotiator* thread.

The *IDSecurityPolicyPPU* class simply implements the abstract methods defined by its super classes. The super PPU classes call upon the *IDSecurityPolicyPPU* to perform specific operations at specific times. The *IDSecurityPolicyPPU* does not control the execution flow of the *PPU* object.

The *IDSecurityPolicyPPU* methods mandated by its super classes perform the policy specific processing. They validate new policy documents, compare policy documents, generate policy negotiation objects and validate policy negotiation objects. All policy documents and policy negotiation objects are encoded with XML and as such these methods typically return an org.w3c.dom.Document object as an output or accept an org.w3c.dom.Document object as an input.

Synchronization

The *IDSecurityPolicyPPU* class is not aware of the multi-threaded environment and does not concern itself synchronization. Its super classes are responsible for applying the locks on shared objects as and when required.

Exceptions

The *IDSecurityPolicyPPU* class throws the *PolicyProcessingException* if encounters any difficulties in performing its policy specific processing.

5.10 Status Code Manager

The *Status Code Manager* maps PBNM status codes to informative strings.

Figure 13 shows the UML class diagram for the *Status Code Manager*. The *EnglishStatusCodeManager* class implements the *StatusCodeManager* interface, which mandates the implementation of the following public methods:

load()	Load the status code translation information.
getStatusString()	Translate a PBNM status code to an informative string.

The load() and getStatusString() methods execute in the context of the caller's thread. These methods, as implemented by the *EnglishStatusCodeManager* class, perform their intended functions without the use of blocking I/O operations and as such they will not cause the caller's thread to block for an extended period of time. However, future implementations of the *StatusCodeManager* interface may choose to load the translation information from files,

directories or databases, which may cause the caller's thread to block waiting for the completion of an I/O operation.

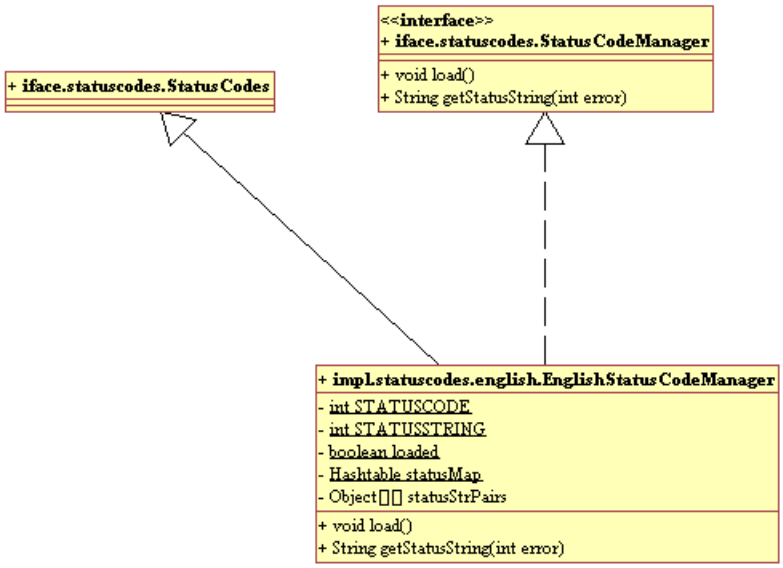

Figure 13 - Status Code Manager UML Diagram

5.10.1 Requirements

The *EnglishStatusCodeManager* must fulfill the following requirements:

1. Loads English language status strings into an internal mapping table indexed by the status code.

2. Maps PBNM status codes to informative strings.

5.10.2 Design

The *Main Program* creates a single instance of the *EnglishStatusCodeManager* class. The public load() method, during its first invocation, loads the English language status strings into a mapping table using the status code as the key and the status string as the value. The public getStatusString() method simply retrieves the status string from the mapping table based on the supplied status code.

5.10.2.1 Synchronization

The public load() and getStatusString() methods are fully synchronized to prevent the concurrent access by the getStatusString() method when the load() method populates the translation table.

5.10.2.2 Exceptions

The *EnglishStatusCodeManager* class does not throw any exceptions.

5.11 Main Program

The *Main Program* creates all the PDP software components and provides references to those objects as requested.

Figure 14 shows the UML class diagram for the *Main Program*. The *Main* class implements the *MainProgram* interface, which mandates the implementation of the following public methods:

getStatusCodeManager()	Provide a reference to the *Status Code Manager*.
getAuthService()	Provide a reference to the *Authorization Service*.
getPolicyProvider()	Provide a reference to the *Policy Info Provider*.
getXMLDsigService()	Provide a reference to the *XML Digital Signature Service*.
getPolicyRepository()	Provide a reference to the *Policy Repository*.
getResolver()	Provide a reference to the *Address Resolution Service*.
getPNPHandler()	Provide a reference to the *PNP Handler*.
getPNPAddress()	Provide the network address for the *PNP Handler*.

These methods execute in the context of the caller's thread. These methods perform their intended functions without the use of blocking I/O operations and as such they will not cause the caller's thread to block for an extended period of time.

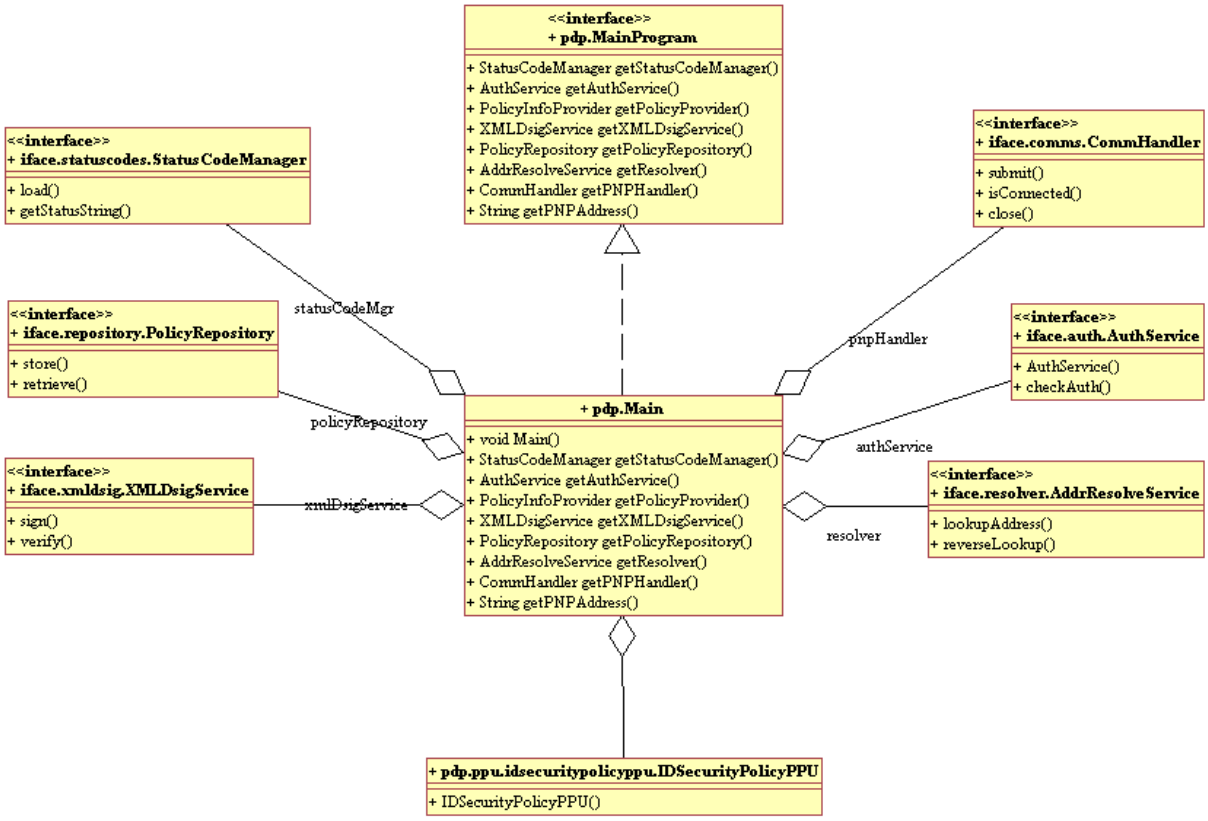

Figure 14 - Main Program UML Diagram

5.11.1 Requirements

The *Main* must fulfill the following requirements:

1. Acquire configuration information from the PDP configuration file.
2. Collect passwords for necessary digital credentials from the Policy Console.
3. Create the necessary PDP software components.
4. Provides references to PDP software component objects as requested.

In the future, the *Main* should use reflection to create the PBNM software components. The class names of these components would instead be specified in the PDP configuration file.

In the future, the *Main* should provide a simple command line interface via the Policy Console.

5.11.2 Design

The *Main* opens the PDP configuration file and retrieves the following configuration items:

- The name of the file containing the digital credentials to be used for PDP digital signatures operations.
- An optional alias associated with the PDP digital credentials.
- The IP address of the XML database server.

- The transport layer port number used by the XML database server.
- The name of the file containing the digital credentials to be used by the *Policy Info Provider*.
- An optional alias associated with the *Policy Info Provider* digital credentials.
- The transport layer port number used by the *Policy Info Provider*.
- The IP address used by the PDP system when communicating with the PNP system.
- The transport layer port number used by the PDP system when communicating with the PNP system.
- The IP address used by the PNP system when communicating with the PDP system.
- The transport layer port number used by the PNP system when communicating with the PDP system.

The *Main* prompts the operator to enter the passwords to unlock the PDP and *Policy Info Provider* digital credentials. The *Main* collects the passwords and creates the PDP software component objects.

The *Main* provides references to PDP software component objects when requested by other PDP software components.

5.11.2.1 Synchronization

The *Main* class does not concern itself with synchronization issues.

5.11.2.2 Exceptions

The *Main* class does not throw any exceptions. However, the *Main* terminates the PDP process (the Java Virtual Machine) if it encounters an error when creating the PDP software components.

References

[RFC2748] RFC 2748, " The COPS (Common Open Policy Service) Protocol ", D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, A. Sastry, January 2000

[RFC3084] RFC 3084, "COPS Usage for Policy Provisioning", K. Chan, J. Seligson, D. Durham, S. Gai, K. McCloghrie, S. Herzog, F. Reichmeyer, R.Yavatkar, A. Smith, March 2001

[IDSP] “Discussion Paper - Specification of Inter-Domain Security Policies”, Version DRAF T 0.3, NRNS Incorporated, December 2004

[IDPE] “Inter-Domain Policy Editor - System Implementation’, Version DRAFT 0.1, NRNS Incorporated, January 2005

[PBNM] “Policy Based Network Management – System Design Document”, Version DRAFT 0.1, NRNS Incorporated, March 2005

[JGROUPS] http://www.jgroups.org

Annex A Sample Trusted Authorities File

The Trusted Authorities file identifies trusted remote certification authorities (CA) by the distinguished name of the CA and the key identifier of the CA's key.

```
<TrustedAuthorities>
    <AD name="AD1">
        <IssuerDN>CN=Root CA,OU=PBNM,O=AD1,C=CA</IssuerDN>
        <IssuerKeyID>-2033668182</IssuerKeyID>
    </AD>
    <AD name="AD2">
        <IssuerDN>CN=Root CA,OU=PBNM,O=AD2,C=CA</IssuerDN>
        <IssuerKeyID>7870182438</IssuerKeyID>
    </AD>
    <AD name="AD3">
        <IssuerDN>CN=Root CA,OU=PBNM,O=AD3,C=CA</IssuerDN>
        <IssuerKeyID>2549814277</IssuerKeyID>
    </AD>
    <AD name="AD4">
        <IssuerDN>CN=Root CA,OU=PBNM,O=AD4,C=CA</IssuerDN>
        <IssuerKeyID>4537265685</IssuerKeyID>
    </AD>
    <AD name="AD5">
        <IssuerDN>CN=Root CA,OU=PBNM,O=AD5,C=CA</IssuerDN>
        <IssuerKeyID>7645639105</IssuerKeyID>
    </AD>
</TrustedAuthorities>
```

Annex B Sample XML Address Resolution Mapping File

The XML Address Resolution Mapping file describes how the local Policy Negotiation Proxy (PNP) communicates with the remote PNP for the associated administrative domain.

```
<AddressResolution>
    <AD name="AD1">
                <Address>10.1.1.1</Address>
                <Port>7100</Port>
                <LocalPort>7101</LocalPort>
        </AD>
    <AD name="AD2">
                <Address>10.2.3.3</Address>
                <Port>7200</Port>
                <LocalPort>7201</LocalPort>
        </AD>
    <AD name="AD3">
                <Address>10.3.3.3</Address>
                <Port>7300</Port>
                <LocalPort>7301</LocalPort>
    </AD>
    <AD name="AD4">
                <Address>10.4.4.4</Address>
                <Port>5800</Port>
                <LocalPort>5801</LocalPort>
        </AD>
    <AD name="AD5">
                <Address>10.5.5.5</Address>
                <Port>5801</Port>
                <LocalPort>5800</LocalPort>
        </AD>
</AddressResolution>
```

Annex C Negotiator State Transition Table

The next two pages contain the state transition table for the *Negotiator* inner class of the *NegotiablePPU*. The current state appears in **blue** in the left column, while the events appear in **blue** in the top row. Actions appear in black and "alert" denotes an alert sent to the *NegotiablePPU*. Finally, the next state in the transition appears in **red.**

Current State	RECHECK Exception	Recheck New Porposal	Recheck Null Proposal	Policy Withdraw	RECV PP w GOOD NT	RECV PP w BAD NT
LOCAL CONFLICT	alert: NEGOTIATION_FAILURE LOCAL CONFLICT	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Do Nothing IDLE	Send PW, alert: END_NEGOTIATION DISENGAGED	Send BAD NT LOCAL CONFLICT	Send BAD NT LOCAL CONFLICT
IDLE	alert: NEGOTIATION_FAILURE LOCAL CONFLICT	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Do Nothing IDLE	Send PW, alert: END_NEGOTIATION DISENGAGED	Send PP, Send GOOD NT, Set a Reminder, alert: NEW_NEGOTIATION WAIT NT	Send PP, Send BAD NT, alert: NEGOTIATION_FAILURE IDLE
WAIT PP	Cancel Reminder, alert: NEGOTIATION_FAILURE LOCAL CONFLICT	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Do Nothing WAIT PP	Send PW, Cancel Reminder, alert: END_NEGOTIATION DISENGAGED	SEND GOOD NT, Set a Reminder WAIT NT	SEND BAD NT, Cancel Reminder, alert: NEGOTIATION_FAILURE IDLE
WAIT NT	Cancel Reminder, alert: NEGOTIATION_FAILURE LOCAL CONFLICT	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Do Nothing WAIT NT	Send PW, Cancel Reminder, alert: END_NEGOTIATION DISENGAGED	Send PP, Send GOOD NT, Set a Reminder, alert: NEW_NEGOTIATION WAIT NT	Send PP, Send BAD NT, Cancel Reminder, alert: NEGOTIATION_FAILURE IDLE
WAIT PR	Send PW, Cancel Reminder, alert: NEGOTIATION_FAILURE LOCAL CONFLICT	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Do Nothing WAIT PR	Send PW, Cancel Reminder, alert: END_NEGOTIATION DISENGAGED	Send PP, Send GOOD NT, Set a Reminder, alert: NEW_NEGOTIATION WAIT NT	Send PP, Send BAD NT, Cancel Reminder, alert: NEGOTIATION_FAILURE IDLE
NEGOTIATED	Send PW, Cancel Reminder, alert: NEGOTIATION_FAILURE LOCAL CONFLICT	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Do Nothing NEGOTIATED	Send PW, Cancel Reminder, alert: END_NEGOTIATION DISENGAGED	Send PP, Send GOOD NT, Set a Reminder, alert: NEW_NEGOTIATION WAIT NT	Send PP, Send BAD NT, Cancel Reminder, alert: NEGOTIATION_FAILURE IDLE
TRY REFRESH	Send PW, Cancel Reminder, alert: NEGOTIATION_FAILURE LOCAL CONFLICT	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Do Nothing TRY REFRESH	Send PW, Cancel Reminder, alert: END_NEGOTIATION DISENGAGED	Send PP, Send GOOD NT, Set a Reminder, alert: NEW_NEGOTIATION WAIT NT	Send PP, Send BAD NT, Cancel Reminder, alert: NEGOTIATION_FAILURE IDLE
REMOTE WITHDRAW	Do nothing, alert: NEGOTIATION_FAILURE REMOTE WITHDRAW	Do nothing REMOTE WITHDRAW	Do Nothing REMOTE WITHDRAW	Send PW, alert: END_NEGOTIATION REMOTE WITHDRAW	Send PP, Send GOOD NT, Set a Reminder, alert: NEW_NEGOTIATION WAIT NT	Send PP, Send BAD NT, alert: NEGOTIATION_FAILURE IDLE
DISENGAGED	Do nothing DISENGAGED	Do nothing DISENGAGED	Do nothing DISENGAGED	Do nothing DISENGAGED	Do nothing DISENGAGED	Do nothing DISENGAGED

Current State	RECV GOOD NT	RCV BAD NT	RECV GOOD PR	RECV BAD PR	RECV GOOD PW	NEG STALLED
LOCAL CONFLICT	Error LOCAL CONFLICT	Error LOCAL CONFLICT	ERROR LOCAL CONFLICT	Send PW LOCAL CONFLICT	alert: END_NEGOTIATION REMOTE WITHDRAW	ERROR LOCAL CONFLICT
IDLE	Error IDLE	Error IDLE	ERROR IDLE	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	alert: END_NEGOTIATION REMOTE WITHDRAW	ERROR IDLE
WAIT PP	Error WAIT PP	Error WAIT PP	ERROR WAIT PP	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Cancel Reminder, alert: END_NEGOTIATION REMOTE WITHDRAW	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP
WAIT NT	Send PR, Set a Reminder WAIT PR	Cancel Reminder, alert: NEGOTIATION_FAILURE IDLE	ERROR WAIT NT	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Cancel Reminder, alert: END_NEGOTIATION REMOTE WITHDRAW	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP
WAIT PR	Error WAIT PR	Error WAIT PR	Set a Reminder, alert: NEGOTIATION_SUCCESS NEGOTIATED	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Cancel Reminder, alert: END_NEGOTIATION REMOTE WITHDRAW	Send PR, Set a Reminder, alert: NEW_NEGOTIATION TRY REFRESH
NEGOTIATED	Error NEGOTIATED	Error NEGOTIATED	Set a Reminder NEGOTIATED	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Cancel Reminder, alert: END_NEGOTIATION REMOTE WITHDRAW	Send PR, Set a Reminder, alert: NEW_NEGOTIATION TRY REFRESH
TRY REFRESH	Error TRY REFRESH	Error TRY REFRESH	Do Nothing, Set a Reminder NEGOTIATED	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	Cancel Reminder, alert: END_NEGOTIATION REMOTE WITHDRAW	ERROR TRY REFRESH
REMOTE WITHDRAW	ERROR REMOTE WITHDRAW	ERROR REMOTE WITHDRAW	ERROR REMOTE WITHDRAW	Send Proposal, Set a Reminder, alert: NEW_NEGOTIATION WAIT PP	ERROR REMOTE WITHDRAW	ERROR REMOTE WITHDRAW
DISENGAGED	Do nothing DISENGAGED	Do nothing DISENGAGED	Do nothing DISENGAGED	Do nothing DISENGAGED	Do nothing DISENGAGED	Do nothing DISENGAGED

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

NRNS Incorporated
4043 Carling Avenue
Ottawa K2K 2A3

2. SECURITY CLASSIFICATION (overall security classification of the document, including special warning terms if applicable)

UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C or U) in parentheses after the title.)

Policy Decision Point (PDP) Software Design Document (U)

4. AUTHORS (Last name, first name, middle initial)

Spagnolo, J., Cayer D.

5. DATE OF PUBLICATION (month and year of publication of document)

September 2005

6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc.)

40

6b. NO. OF REFS (total cited in document)

6

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Contract Report

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include the address.)

DRDC Ottawa/NIO Section
3701 Carling Avenue
Ottawa K1A 0Z4

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant)

15BF27

9b. CONTRACT NO. (if appropriate, the applicable number under which the document was written)

W7714-3-800/001/SV

10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document.)

10b. OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor)

DRDC Ottawa CR 2005-110

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(x) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; further distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

Full Unlimited

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

(U) This document presents a software design for the Policy Decision Point (PDP) component of a policy-based network management (PBNM) system. The PDP facilitates the compilation of policies, the storage of policies, the exchange of policies, the evaluation and negotiation of policies, as well as the implementation and enforcement of policies.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Inter-domain security policy
Network Managament
Policy
Policy-based network management (PBNM)
Policy Decision Point (PDP)
Policy enforcement
Policy Enforcement Point (PEP)
Policy negotiation
Policy Negotiation Proxy (PNP)
Policy object
Policy Processing Unit (PPU)
Security Policy
XML Policy

Defence R&D Canada

Canada's leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

www.drdc-rddc.gc.ca