# A Java API for X.509 Proxy Certificates

John Gilbert, Russell Perry
HP Laboratories
HPL-2008-77

**Keyword(s):**
X.509 Proxy Certificate, Delegation, Public Key Infrastructure, Grid Security Infrastructure, SSL, HTTPS, Java Cryptography Architecture, Java Secure Sockets Extension.

**Abstract:**
X.509 Proxy Certificates have been proposed for use in the Grid Security Infrastructure to allow dynamic delegation of rights and single sign-on for end users. We have evaluated proxy certificates to secure a service-oriented architecture for digital content based on Web Services. We describe how support for proxy certificates was implemented in Java through extensions to the Java Cryptography API and related security APIs. The principal challenges involved providing control over which proxy certificate to use per SSL connection, validating proxy certificate chains and supporting runtime generation of proxy certificates.

External Posting Date: July 6, 2008 [Fulltext] Approved for External Publication

Internal Posting Date: July 6, 2008 [Fulltext]

# A Java API for X.509 Proxy Certificates

John Gilbert (john.gilbert@hp.com)[1]
Russell Perry (russell.perry@hp.com)

Hewlett Packard Laboratories
Bristol, UK

### Abstract

X.509 Proxy Certificates have been proposed for use in the Grid Security Infrastructure to allow dynamic delegation of rights and single sign-on for end users. We have evaluated proxy certificates to secure a service-oriented architecture for digital content based on Web Services. We describe how support for proxy certificates was implemented in Java through extensions to the Java Cryptography API and related security APIs. The principal challenges involved providing control over which proxy certificate to use per SSL connection, validating proxy certificate chains and supporting runtime generation of proxy certificates.

### Keywords

X.509 Proxy Certificate, Delegation, Public Key Infrastructure, Grid Security Infrastructure, SSL, HTTPS, Java Cryptography Architecture, Java Secure Sockets Extension.

## 1 Introduction

X.509 Proxy Certificates have been proposed by the Grid Computing community for use in the Grid Security Infrastructure (GSI) allowing delegation of rights and single sign-on for end users [1]. A proxy certificate is a certificate that is derived from and signed by a normal X.509 Public Key End Entity Certificate (EEC) or by another proxy certificate. They are realized as regular X.509 Public Key Certificates [1] with an embedded proxy certificate information extension containing proxy certificate specific information. The structure of the extension, an associated application profile for its use and validation requirements are defined by the PKIX working group. At the time of performing this work[1], the format of the information extension had just reached RFC status within the IETF [3].

We assessed proxy certificates for use in the security infrastructure of a service-oriented architecture for digital content based on Web Services. This required the development of a Java API to allow developers to easily add proxy certificate support to secure existing Web Services. To achieve this, a proxy certificate aware SSL implementation was

[1] The work was performed over the period 2004-05.

developed with additional support for using HTTPS in conjunction with proxy certificates. The implementation was created as a set of extensions to the Java Cryptography API and related APIs; the version of JDK was 1.4, although the approach should be applicable to later versions. The implementation involved tackling several challenges which are detailed in section 3 and which comprises the main part of this report.

The report is organized as follows. Section 2 provides an overview of proxy certificates and their applicability. Section 3 contains the detailed description of how the proxy certificates were implemented using the Java Cryptography API and related security APIs. A summary is presented in Section 4. Additional information about the API is provided in the Appendix.

# 2 Introduction to Proxy Certificates

The use of proxy credentials is a relatively new technique introduced in [1]. Two uses of proxy certificates of interest to out service-oriented architecture are enabling *single sign-on* and *delegation* over a network.

*Single sign-on* requires a user to authenticate only once, within a domain, using their X.509 certificate in order to create a proxy certificate. Thereafter, the proxy certificate can be used to authenticate with multiple services within the domain. *Delegation* entails a user, granting some or all of its privileges to another entity to perform some action on its behalf that involves calling on other services that may enforce some form of access control. For example, in our context a user may request a format transcoding service to transform content held in a separate content management system; the transcoding service should have access limited by the user's privileges and conversely the user should not be able to gain access to content indirectly for which they do not have permissions.

The following two sub sections provide an outline of how proxy certificates are used to support these usage patterns.

## 2.1 Single Sign-On

Typically, a user's long term Private Key is protected in some manner that requires them to manually authenticate each time access to it is required e.g. the key is stored in an encrypted format on a smart card. This introduces a large amount of overhead if the user is required to access this key to authenticate with other entities frequently, although it provides a high level of protection for the key. It is possible to cache the key on the local system, removing the problem of repeat-access to the secured key, although this increases the possibility of the key becoming compromised. An alternative approach allows the creation of a short lived proxy certificate, signed using the long term Private Key, which can be cached on the local system. This is illustrated in Figure 1.

Given the short lifespan of the proxy certificate, fewer security mechanisms need be employed for its protection than for the long term credentials – those of a computer file

system are often sufficient. This proxy certificate and associated Private Key may then be used without requiring the user to repeatedly authenticate manually. Note, only in this use of proxy certificates is the private key of the proxy certificate known to the user.

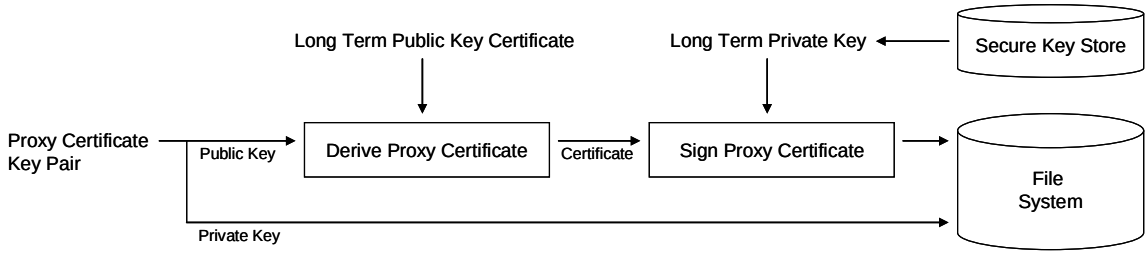


Figure 1: Proxy certificate creation for single sign-on

## 2.2 Delegation over a network

Delegation over a network is possible between two processes without exchanging private keys. After mutual authentication has been performed using end entity certificates, an integrity protected channel is established over which to communicate. This is typically performed using SSL, although the privacy offered by SSL is not strictly required. The creation of a proxy certificate is illustrated in Figure 2. In this example, Host A invokes a service on Host B, which acts on Host A's behalf to call the service on Host C. Host B generates a pair of public/private keys and forms a request for a proxy certificate which is sent to Host A over the integrity protected channel. Host A derives a proxy certificate as described above for single sign-on, which is signed using its private key and returned to Host B. Host B can then use this certificate in association with the newly created private key to create connections on Host A's behalf without ever compromising Host A's private key, or indeed the private key associated with the proxy certificate. In this usage, the initiator, Host A, does not have access to the private key of the proxy certificate.

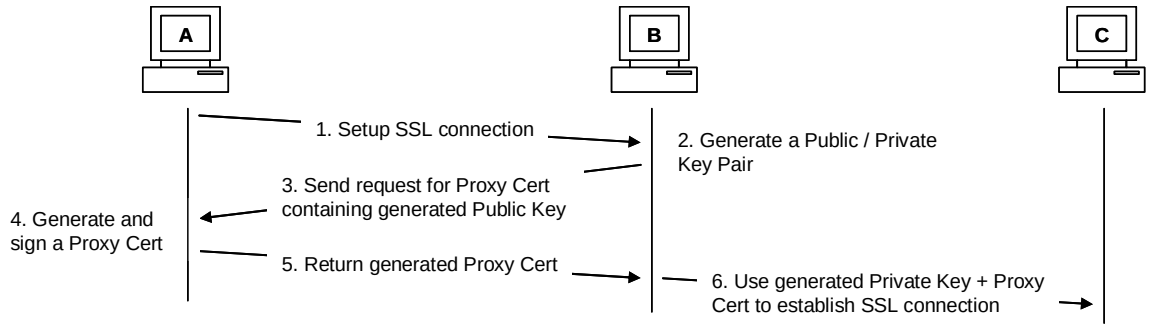


Figure 2: Delegation using a Proxy Certificate

In the implementation described in this report, the originator delegates full rights to the bearer of the certificate. Thus, services presented with the proxy certificate will treat the bearer as they would the issuer. However, proxy certificates have unique names derived from the name of the certificate from which they are issued, and as such it is also possible for them to have their own rights independent of their issuers.

## 2.3 Structure of a Proxy Certificate

Part of the security infrastructure supporting X.509 certificates is the Certificate Authority (CA). The CA is a trusted third party that issues certificates binding a Public Key to a particular name. This name is called a Distinguished Name (DN), whose format is defined by the X.500 Directory Services system. The structure of, and information contained in, an X.509 Public Key certificate is shown in Figure 3.

A proxy certificate is derived as an extension to a Public Key certificate, using the proxy certificate information extension defined in RFC3820 [3]. The presence of this extension in a certificate indicates that the certificate is a proxy certificate, and contains information on whether the issuer of the certificate has placed any restrictions on its use. The structure of the extension is shown in Figure 4.

The Path Length Constraint field specifies the maximum depth of the path of proxy certificates that may be signed by this proxy certificate if present. If this field is set to 0 the certificate must not be used to sign a proxy certificate, however if it is not specified the depth is unrestricted.

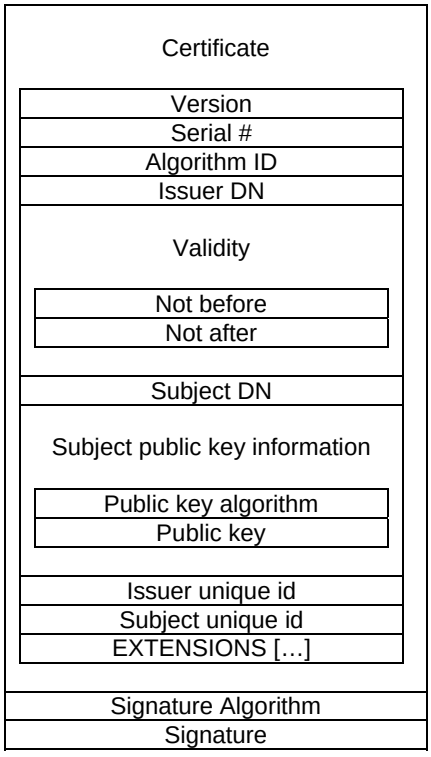


Figure 3: Structure of an X.509 Public Key Certificate

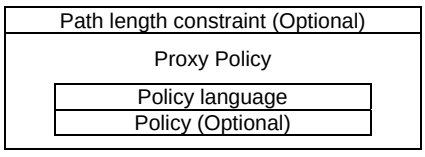


Figure 4: Proxy Certificate Information Extension

The Proxy Policy field specifies a policy on the use of this certificate for the purpose of authorization. The Policy Language field specifies the format (an OID uniquely defining a policy language known to both parties) of the policy. The RFC does not define any particular language for use in policy specification, leaving this task to application developers. However, two important values for the language field are defined which must be understood. These are *id-ppl-inheritAll* which indicates that simple unrestricted proxying is in use and *id-ppl-independent* which indicates that this certificate inherits no rights from the issuer and thus the only rights are those that are explicitly granted to it. All other values indicate another policy and associated language is in use, optionally making use of the policy field to express that policy.

An associated application profile is included in the specification which sets out how X.509 Public Key Certificates are used as proxy certificates. Restrictions include

- The proxy certificate information extension must be present and must be marked as a critical extension so that applications that are unable to process the extension will not validate and accept the certificate.
- The Issuer of a proxy certificate may be either an end entity certificate (i.e. a certificate signed by a CA), or another proxy certificate, whose Subject must not be empty.
- The Issuer field in a proxy certificate must contain the Subject field from the certificate that issued the proxy certificate.
- Issuer and Subject alternative names must not be present in a proxy certificate.
- The Subject field of a proxy certificate must be the Issuer field (i.e. the subject of the certificate that issued the proxy certificate) with a single additional unique Common Name (CN) component. In this way all Subject names in proxy certificates are derived from the name of the issuing end entity certificate (EEC).

A number of other subtle restrictions are placed on the use of fields in the certificate which are specified in [3].

Once validated, a path of certificates can bind a public key to a subject, and ensure that this subject has indeed been delegated to by those certificates which precede it in the chain.

## 2.4 Existing implementations

We required a Java implementation for proxy certificates which we could easily integrate into our service-oriented architecture. At the time, a number of implementations already existed for processing proxy certificates as part of grid middleware and other applications, a few of which are described below.

- The Globus Toolkit [5] upon which grid applications can be built provides services and libraries for resource monitoring, discovery, management and security. The Grid Security Infrastructure (GSI) forms part of this toolkit, and provides libraries for working with proxy certificates. These were implemented

in C, although the Java CoG Kit [6] provides access to the Globus grid services through the Java framework – including the GSI.

- GridSite [7], a set of extensions to the Apache web server that allow the modifications to a website based on the authentication of a users web browser using X.509 Public Key Certificates or proxy certificates. It is written in C, with custom OpenSSL [8] validation routines built into its system. The implementation specifically supported Globus GSI proxy certificates (from which the IETF specification arose), attempting to accept both old style (where only CN=proxy or CN=limited proxy were valid additions to the issuers DN) and new style (allowing arbitrary naming as long as it's scoped to the issuers name) certificate formats.

- The European Datagrid Java Security project [9] implemented a Java TrustManager to allow Tomcat make use of Globus proxy certificates (CN=proxy/CN=limited proxy) for setting up SSL, however the CVS repository was no longer available and their implementation for authentication was based on the draft-ietf-pkix-proxy-05.txt document rather than the current specification.

Few of these implementations followed the guidelines in detail set out in RFC 3820 – given its publication date of 2004. Most were not Java based, and of those that were they either suffered from the aforementioned problem, or were not in a form suitable to integrate into our service-oriented architecture.

# 3 Implementing Proxy Certificates using the Java Cryptography Architecture

One of the key objectives of this work was to develop a JCA provider to allow a third party developer to readily integrate proxy certificates in a service-oriented architecture that used HTTP(S) as the primary communication protocol. Additionally the proxy certificates were to be created during run time. A primary concern with the API design was that it closely resemble and be compatible with the existing Sun SSL Sockets and URL APIs, reducing the need for developers to familiarize themselves with a new set of interfaces and concepts.

To this end, a JCA Provider [10] was implemented (`com.hp.hpl.csf.security`) which offers the services required to set up raw SSL connections using proxy certificates. Additionally, a URL handler for the HTTPS scheme was implemented (`com.hp.hpl.csf.security.https`) to allow a client application to connect to a proxy certificate aware HTTP(S) Server. Finally, server side handling of proxy certificates was implemented using the HTTP Server provided by MortBay in the package (`com.hp.hpl.csf.security.mortbay`) [12]. The appendix summarizes the classes and interfaces that have been implemented as well as other supporting material such as sample configuration files for Jetty.

Creating the JCA provider presented numerous challenges. These are summarised below and are addressed in the sub-sections indicated.

1. An interface is required which allows the creation of an `SSLSocket` using a *named certificate* sourced from a `KeyManager`. This is not supported in the default `SSLSocketFactory`. This challenge is addressed in section 3.2.

2. Support is required for the *creation*, *issuance* and *validation* of proxy certificates. During runtime new proxy certificates will need to be created and issued. The actual mechanism for this is not specified in [3]. Therefore, additional utility code to support proxy certificate issuance is helpful to assist the developer. The `TrustManager` must also be able to support validation and authentication of a proxy certificate chain. These challenges are addressed in section 3.3.

3. In the default initialization of the `SSLContext` and related classes, changes to the `KeyStore` are not propagated to dependent classes. Thus to support for dynamically issued proxy certificates stored in the `KeyStore` at runtime, dependent classes such as the `TrustManager` and `KeyManager` will need to be notified and refreshed to maintain consistency with the `KeyStore`. This challenge is addressed in section 3.4.

Before addressing each of the challenges listed above, an overview of the Java Cryptography Architecture and extensions is presented in section 3.1.

## 3.1 The Java Cryptography Architecture and relevant APIs

The Java Cryptography Architecture (JCA) is a framework for accessing and developing cryptography functionality for the Java platform [10]. It was introduced in the first release of the Java Security API, a core API in the Java programming language that allows developers to add both low and high level security functionality to their software. There are three related APIs that are relevant to implementing proxy certificates. These are:

1. The *Java Cryptography Extension* (JCE) extends the JCA to provide a framework for the development of ciphers and includes default implementations.
2. The *Java Secure Sockets Extensions* (JSSE) API supplements the core cryptographic services by providing extended networking socket classes to developers that offer authentication, encryption and integrity protection at the transport level via SSL or TLS.
3. The *Certification Paths API* provides classes and interfaces for handling ordered lists of certificates called certification paths (also known as certificate chains), which may be used to securely establish the mapping of a public key to a subject if they meet certain validation requirements.

Prior to Java 2, the JCE and JSSE APIs were optional packages which could not be packaged as standard in Java due to US export regulations, however both are now integrated and shipped with Java.

All the APIs (JCA, JCE, JSSE and Certification Paths) are extensible and follow a design pattern originally specified for extending the core JCA, allowing implementation independence by the use of a *Provider* based architecture. A package that implements one or more Cryptographic Services is termed a Cryptographic Service Provider ('Provider' for short).

Developers request a particular type of object implementing a particular service (e.g. A key generator that generates RSA keys), or alternatively request an implementation from a specific provider if required (e.g. Sun's implementation of a key store). An *engine class* defines a cryptographic service in an abstract way and is always associated with a particular algorithm. Its methods are all declared final. The application interface defined by an engine class is implemented by a Service Provider Interface (SPI). For each engine class there is a corresponding SPI class, which is sub classed to implement a particular cryptographic service. An instance of an engine class encapsulates an instance of a corresponding SPI-derived class. A group of cryptographic services are registered with the JCA via a `java.security.Provider` implementation, which specifies which services are implemented and which classes provide those services. Providers can be registered with the JCA statically via configuration in the `java.security` file, or dynamically at run time.

The typical steps involved in using JCA to create a secure socket are illustrated in Figure 5. The steps are as follows

1. Create an instance of `KeyStore` and load a collection of trusted Public Key certificates and <Private Key, Public Key Certificate> pairs from a file.

2. Create `TrustManagerFactory` and `KeyManagerFactory` objects which are initialized with the `KeyStore` from the previous step. These factories will return an array of `TrustManager` and `KeyManager` objects respectively for each distinct type of public key certificate and <Private Key, Public Key Certificate> pair held in the `KeyStore` (e.g. X.509).

3. Initialize an `SSLContext` using the arrays obtained in the previous step. Using the `SSLContext` obtain `SSLSocketFactory` and `SSLServerSocketFactory` objects.

4. Use the `SocketFactory` objects to create `SSLSocket` and `SSLServerSocket` objects respectively.

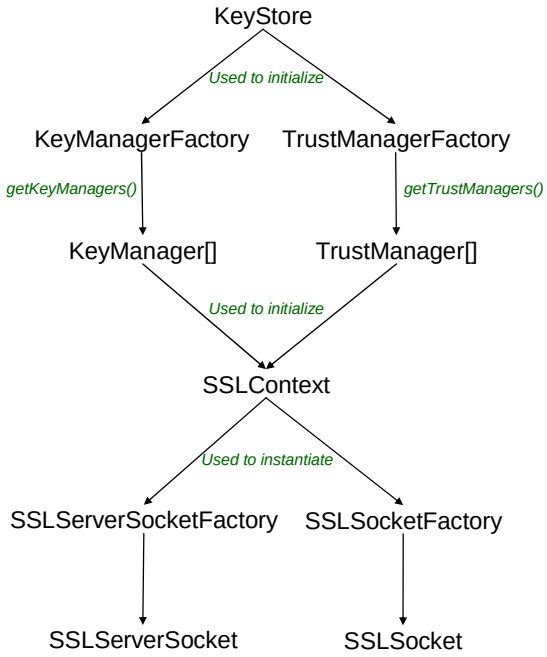


Figure 5: Classes used to create a secure socket

Although not directly related to JCA, Java's URL framework is also relevant to our aim of supporting a new HTTPS protocol handler that supports specification of named certificates and proxy certificate verification. The URL framework has been designed to allow developers to easily implement custom protocol handlers to replace the JDK's default handlers. For our purposes, we need to provide an alternative handler for the HTTPS scheme. A full description of the framework is beyond the scope of this document. However, we note a number of naming conventions which must be adhered to:

1. Classes to implement a custom protocol handler must be put into a package whose name ends in the scheme of the protocol which it implements. E.g. the package `com.hp.hpl.csf.security.https` will contain classes for handling URLs with the HTTPS scheme.

2. A subclass of `URLStreamHandler` named `Handler` must reside in the package implementing the single method

```
protected URLConnection openConnection(URL url)
```

The URL package is registered with the Java runtime by setting the `java.protocol.handler.pkgs` property to point at the package containing the protocol implementations (e.g. `com.hp.hpl.csf.security`). Whenever a call to `URL.openConnection()` is made, the `openConnection(url)` method will be invoked for all URLs with scheme matching that specified in the package name; in this case https.

## 3.2 Creating a secure socket using a named certificate

An SSL handshake is an exchange of messages which occurs at the start of every SSL session during which a client and server negotiate certificates that are acceptable to both for authentication. This aspect of the negotiation is supported in the `KeyManager` interface by the `chooseClientAlias` and `chooseServerAlias` methods to which callbacks are made from within the JSSE when a certificate selection is required.

In our context, the information provided from the server, to the client does not provide necessary context for local certificate selection. For example, referring to Figure 2, Host B would typically receive requests from multiple clients and thus would cache multiple proxy certificates. Also in our service-oriented architecture, proxy certificates are to be created at runtime with relatively short lifetimes and/or different rights granted. Therefore, the developer must be able to implement control over which proxy certificate is used to establish an SSL connection.

Unfortunately, the existing `SSLSocketFactory` does not allow the creation of an `SSLSocket` using a named certificate. Thus some mechanism to allow a named certificate on the client side to be selected for authentication per connection needs to be exposed, rather than allowing an arbitrary certificate selection occur. This is illustrated in Figure 6.

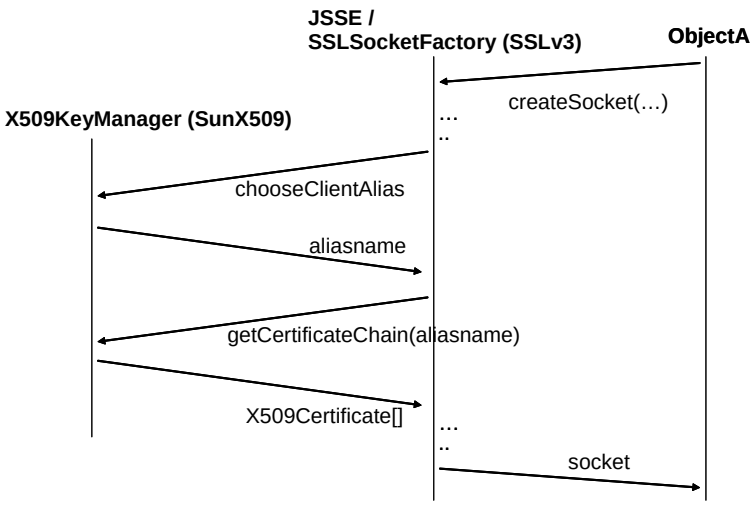


Figure 6: Interaction between an SSLSocketFactory and an X509KeyManager during SSL handshake (client side)

This limitation was overcome by implementing a custom class (`PCKeyManager`) implementing the `X509KeyManager` interface and an associated `KeyManagerFactorySpi` (`PCKeyManagerFactory`) for `PCKeyManager` creation. Essentially the `PCKeyManager` uses the delegation pattern and encapsulates an instance

of X509KeyManager. The PCKeyManager augments the X509KeyManager interface with an additional method setClientAlias for the purpose of named-certificate SSLSocket creation, which allows the setting of a private member called activeAlias. If no alias is specified, all method calls are delegated to the encapsulated KeyManager. If the activeAlias member is specified at the time a call is made to chooseClientAlias, the value of activeAlias is returned instead.

At this stage, if the PCKeyManager were to be used to initialize an SSLContext *using a named certificate*, from which an SSLSocketFactory is then obtained, a lock would need to be held on the PCKeyManager instance (e.g. aPCKeyManager) for the duration of the SSL handshake as illustrated in the code fragment below (error handling removed for clarity)

```
synchronized (aPCKeyManager) {
        aPCKeyManager.setActiveAlias(alias);
        socket = (SSLSocket) sf.createSocket(...);
        socket.startHandshake();
        aPCKeyManager.setActiveAlias(null);
}
```

By holding the lock on the PCKeyManager instance aPCKeyManager, the alias cannot be changed whilst the SSL handshake is being performed thus guaranteeing that the correct proxy certificate is used. While this approach certainly gives the desired effect, it is very fragile and prone to developer error. For this reason a new SSLSocketFactory class was implemented called PCSSLSocketFactory, which delegates to the default SSLSocketFactory whilst extending the interface with a number of additional methods. Specifically new overloaded methods are provided which extend each existing createSocket method signature from SSLSocketFactory with the addition of a string parameter to specify the alias to be used for certificate selection. Within these signature-extended methods, code fragments similar to the above are encapsulated, thus allowing the developer to specify the alias to create a socket without concerning themselves with thread safety. All other interface methods are implemented by delegation to the encapsulated SSLSocketFactory. Note it's possible to further extend PCSSLSocketFactory, to implement the HandshakeCompletedListener interface to receive callbacks from a SSLSocket after the SSL handshake is completed.

The two figures below illustrate the sequence of interactions between the classes PCKeyManager and PCSSLSocketFactory described above and a calling object denoted ObjectA when an alias is not specified (Figure 7) and when it is (Figure 8).

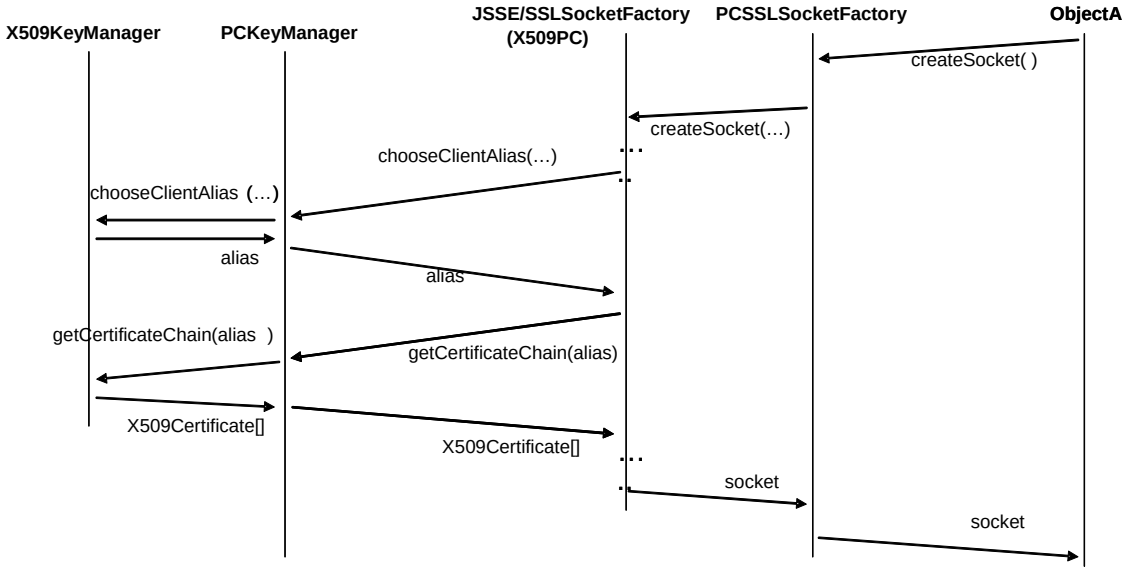


Figure 7: Sequence of method calls on client to setup an SSL connection without specifying the alias

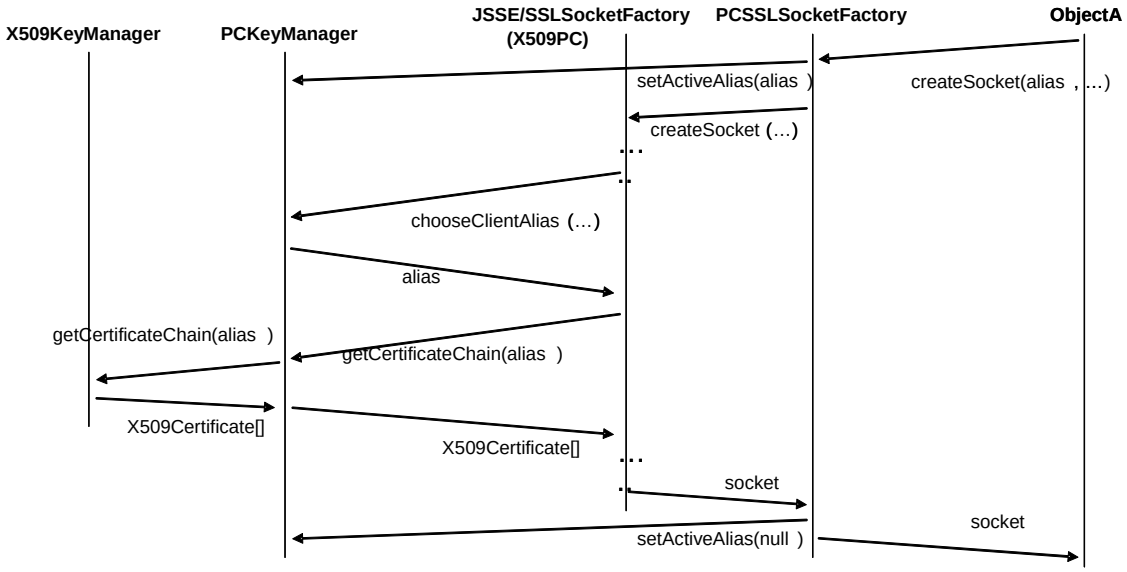


Figure 8: Sequence of method calls on client to setup a named certificate SSL connection

A complication to the sequences shown in Figure 7 and Figure 8 arises because the SSL specification allows for the resumption of sessions. Specifically in Sun's implementation of the `createSocket` method on `SSLSocketFactory`, an entire SSL handshake is only performed for each distinct set of parameter values. The underlying `SSLContext` from which an `SSLSocketFactory` is instantiated simply 'resumes' a previous connection if the parameters match those used in a previous connection. Whilst this is generally desirable given the overhead of an SSL handshake, it prevents the user from establishing connections to the same destination using different certificates, hence authenticating with different identities and/or different delegated rights.

To avoid this, a new `SSLContext` is instantiated for each connection that uses a proxy certificate for authentication. Within the `PCSSLSocketFactory` implementation we store the necessary state to perform this action, whilst encapsulating an instance of the default `SSLSocketFactory` to delegate `createSocket` calls that do not specify an alias identity.

In order to provide the same interface for the creation of `PCSSLSocketFactory` objects as for regular `SSLSocketFactory` objects it is necessary to create a custom `SSLContext` by implementing an `SSLContextSpi` (`PCSSLContext`). This context creates `PCSSLSocketFactory` objects and returns them when `getSocketFactory` is invoked. Unfortunately, due to export regulations the JDK 1.4.x release did not allow the return of custom `SSLSocketFactory` or `SSLServerSocketFactory` objects from an `SSLContext`. Thus developers wishing to use the functionality provided by `PCSSLSocketFactory` objects must directly instantiate them rather than create them through an `SSLContext`.

## 3.3 Proxy Certificate Creation, Issuance and Validation

The next step to support developers using proxy certificates, is to provide a means to create new proxy certificates, request their issuance (i.e. certificate request) and validate a certificate path containing proxy certificates. These steps are described in this section.

### 3.3.1 Creation

To simplify the creation of X.509 proxy certificates, the Bouncy Castle [14] `X509V3CertificateGenerator` was extended with methods to set and get the values of the fields defined in the proxy certificate information extension [3]. Refer to Figure 4. The extra methods are

- `getPathLengthConstraint`
- `setPathLengthConstraint`
- `getPolicyLanguage`
- `setPolicyLanguage`
- `getPolicy`
- `setPolicy`

A utility method `updateProxyExtension` forms an ASN.1 DER encoding of the information which has been specified using the methods above and adds it to the set of certificate extensions. The `generateX509Certificate` methods are overridden, calling `updateProxyExtension` before generating the certificate, and then wrapping the generated certificate in a `PCertificate` before returning it to the calling user.

### 3.3.2 Issuance

When the user invokes an operation on a service, the service may require a proxy certificate to be issued by the user in order to perform the requested operation. This is because services it calls on behalf of the user enforce access control policies based on the end user's identity, or assigned roles, and not the identity of the service making the call. The service infrastructure thus needs to define

1. Metadata describing which, if any, of the operations offered by a service must be invoked over an SSL connection established with a proxy certificate issued by the user. In a more sophisticated model, the metadata may also describe the rights the service requires in order to fulfill a given operation. Various Web Service specifications address this issue.
2. A means for the user to first connect to a service and a protocol which defines (i) how the user should request a service to generate a new key pair for a proxy certificate and secondly (ii) how the service will send a signing request to the user using the generated public key.
3. An optional policy language for the user to specify rights which can be delegated.

In our service architecture, the intent was to be able to trace back a request to the identity of the originating user, the so called *identity based authorization with impersonation* [1]. In effect, the user delegates all rights to a service they call. Therefore, there is no need for the user to place restrictions on the rights granted to the service. To further simplify, it is assumed here that a services must always be invoked by the user using a proxy certificate issued beforehand by first establishing a secure socket using end entity certificates. The steps involved in issuance, creation and subsequent use of a proxy certificate are illustrated in Figure 9.

Referring to Figure 9, the user (not necessarily human) first connects to Service A, and establishes a secure connection between both parties using their end entity certificates (step 1). The user is then assumed to request the issuance of a new public private key pair as the basis for a new proxy certificate using the secure connection. The syntax of the issuance request is undefined, but could be simply realized as a RESTful service call. The target service creates the public and private key pair (step 2) and sends the public key to the user in a signed certificate request over the secure channel (step 3). The user then signs the certificate request with their private key associated with their end entity certificate thereby generating a new proxy certificate which is sent to Service-A (step 4). The user may then invoke operations on Service A which may then use the user's proxy certificate to call service-B (step 5). In so doing Service-B (steps 6, 7) is now able to attribute the request from the originating user. Note that Service A will in practice deal with multiple requests in parallel from different users. This is why control over which alias/certificate is used to establish a secure connection (e.g. to service B) is required

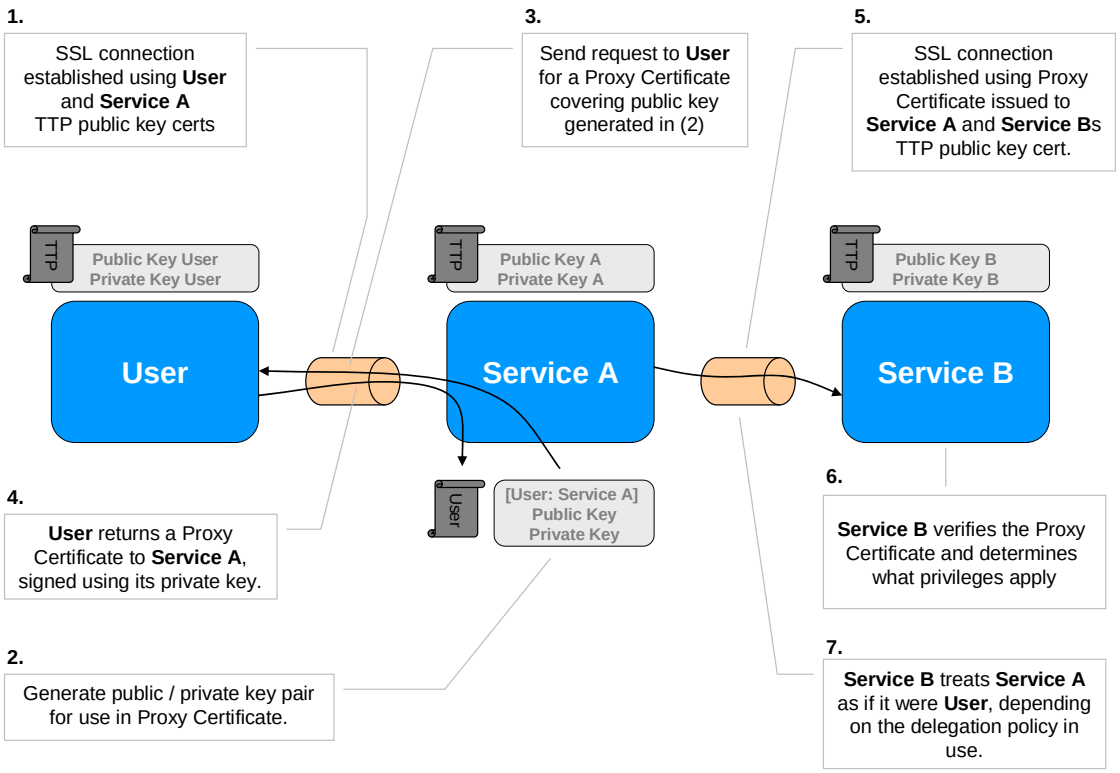


Figure 9: Process for requesting issuance and creation of a proxy certificate

### 3.3.3 Validation

The standard method for validating an X.509 certificate chain is to validate the path from an end-entity certificate back to a trusted Certificate Authority (CA). The certificate path is an ordered list of certificates where each certificate is issued by the next in the chain. However, the standard PKIX algorithm for such validation does not apply to chains that contain proxy certificates issued by EECs rather than by CA certificates. Fortunately, the Certification Paths API provides a set of interfaces which allow users to either use existing algorithms for validation, or implement their own validation algorithms. Therefore, an algorithm was implemented to validate certificate chains containing proxy certificates in `PCPathValidator` a concrete instance of `CertPathValidatorSpi`.

This `PCPathValidator` class first validates the section of the chain from the first non-proxy certificate to the end of the chain using the standard PKIX algorithm, and following this validates from the first non-proxy certificate back to the first proxy certificate iteratively. The implementation follows the algorithm specified in RFC 3820, however it is left to programmers using the API to extract the appropriate policy

information from a validated Certificate Path. Section 4.1.3 (d) from the RFC concerning processing of other certificate extensions is not currently implemented.

In order to support X.509 proxy certificates in the validation chain, the `PCPathValidator` introduced above was used in a custom `X509TrustManager` called `PCTrustManager` to determine whether or not a given certificate chain is trusted. At the server, the `checkClientTrusted` method is called to validate credentials provided by the client to the server using the `PCPathValidator` just described. Clients determine whether they should trust a server by calling the `checkServerTrusted` method (our current implementation requires a certificate chain pass regular PKIX validation with anchorage at a trusted certificate). Like the implementation of `PCKeyManager`, the `PCTrustManager` instance encapsulates the default `X509TrustManager`, to which `getAcceptedIssuers` invocations are delegated. An associated `TrustManagerFactorySpi` was implemented called `PCTrustManagerFactory` for the creation of `PCTrustManagers`.

## 3.4 Propagation of KeyStore updates at runtime

The next significant challenge was to allow for proxy certificates to be created by the user during runtime and to then make them immediately available for use to establish new secure connections to services. To allow for this, the `KeyStore` must be able to propagate changes during runtime.

Unfortunately, there is a cascading initialization of all objects involved in the setup required for establishing SSL connections, originating at the `KeyStore` (see Figure 5). The implementation of the default classes that ship with Suns JSSE will not propagate updates from the `KeyStore` through to the `SSLContext` without reinitializing each object in the chain manually. This occurs because each initialization step shown in Figure 5 occurs by value using a cloned object rather than by reference. Thus, while a given certificate might have been inserted into the underlying `KeyStore` it will not be available for authentication of SSL connections until all objects through to the `SSLContext` have been re-initialized. In our service architecture, we require updates to the `KeyStore` be immediately reflected in associated `SSLContext` instances. In this way, new SSL sockets can be created using recently generated proxy certificates.

Two capabilities must be added to support dynamic updates. Firstly, detecting when the `KeyStore` is updated and secondly, propagating updates to previously initialized objects.

To address the first problem, Sun's `KeyStore` was encapsulated in a custom `KeyStoreSpi` implementation called `PCKeyStore` which delegates calls to the encapsulated `KeyStore`, but adds additional code to those methods which cause a change in state of the `KeyStore` to notify listeners of changes. The relevant methods are `load`, `setCertificateEntry` and `setKeyEntry`.

In addressing the second problem, either a *push* model where an object actively informs other registered listeners objects of updates, or a *pull* model where the objects poll objects to determine if it has been updated. Our implementation uses the push approach to propagate `KeyStore` changes i.e. the `KeyStore` notifies the related SPI objects of changes to its contents. There are however a couple of issues that complicate this approach.

1. The JCA *engine class* [10] which encapsulates the concrete Service Provider Interface (SPI) implementations of both `TrustManagerFactory` and `KeyManagerFactory` objects and critically `KeyStore` objects cannot be modified to support notifications. Ideally, `KeyStore` updates could be sent directly to the encapsulated SPI implementation, but direct access through existing APIs is not possible.
2. A means to reinitialize the `PCTrustManager` and `PCKeyManager` instances referenced by the `SSLContext` is required (refer to Figure 5). This is to allow updates to the `KeyStore` to be reflected in the current `PCTrustManager` and `PCKeyManager` instances.

To tackle the first issue, a singleton class `PCKeyStoreUpdateBroadcaster` was introduced with which objects implementing the `PCKeyStoreUpdateListener` interface can register to receive notifications of `KeyStore` changes. Additionally, custom `PCKeyManagerFactory` and `PCTrustManagerFactory` implementations were created which implement the `KeyStoreUpdateListener` interface and register themselves with the `KeyStoreUpdateBroadcaster`. In effect, this registers the SPI implementation rather than the external Engine. When a successful update operation occurs on the `PCKeyStore`, it calls the `informRegisteredUpdateListeners` method on the `KeyStoreUpdateBroadcaster` to notify the `PCKeyManagerFactory` and `PCTrustManagerFactory` of the update.

As soon as the `PCKeyManagerFactory` and `PCTrustManagerFactory` instances are notified of the `KeyStore` update they must reinitialize the associated `PCTrustManager` and `PCKeyManager` instances for the update to take effect. This is the second issue identified above and is addressed by exploiting the extra level of indirection provided by the delegation pattern used extensively in this work. Specifically, the `PCKeyManagerFactory` class encapsulates an underlying `KeyManagerFactory`. A call to the `getKeyManagers` method on `PCKeyManagerFactory` is first delegated to the `KeyManagerFactory`. Each returned `KeyManager` is encapsulated in an instance of a `PCKeyManager` which is then returned and used to initialize an `SSLContext`. A similar delegation pattern is implemented in the `PCTrustManagerFactory`. The `PCKeyManagerFactory` and `PCTrustManagerFactory` also cache their initialization parameters which include the `KeyStore`. Whenever the `KeyStore` is updated, the encapsulated `KeyManager` and `TrustManager` instances of the `PCKeyManager` and `PCTrustManager` are updated by the `PCKeyManagerFactory` and `PCTrustManagerFactory` respectively using the `setHandleRef` methods on

PCKeyManager and PCTrustManager. Since the SSLContext is initialized from the PCKeyManager and PCTrustManager instances, it is effectively updated i.e. new SSL connections can be made referring to certificates added to the KeyStore after the SSLContext was initialized. Essentially the delegation pattern provides a stable PCKeyManager and PCTrustManager reference for the SSLContext, whilst allowing the encapsulated KeyManager and TrustManager instances to be updated as necessary.

In summary, the propagation of PCKeyStore updates occurs as follows:

1. A PCKeyStore is created, wrapping another KeyStore.
2. PCKeyManagerFactory and PCTrustManagerFactory instances are created and initialized using the PCKeyStore from (1). These Factory instances register themselves with the singleton PCKeyStoreUpdateBroadcaster. Each instantiates the associated JDK Factory object and initializes it using the PCKeyStore. X509KeyManager and X509TrustManager instances are then obtained from their associated factory classes and encapsulated in PCKeyManager and PCTrustManager instances by the PCKeyManagerFactory and PCTrustManagerFactory respectively.
3. Calls to getTrustManagers / getKeyManagers on those custom Factories from (2) return the persistent PCKeyManager and PCTrustManager instances inside arrays. These arrays are used to initialize an SSLContext.
4. When an update operation occurs on the PCKeyStore (e.g. a newly issued proxy certificate is added), it calls the informRegisteredUpdateListeners method on the singleton PCKeyStoreUpdateBroadcaster. This iterates over all registered PCKeyStoreUpdateListener objects, invoking their keyStoreUpdated method.
5. In response to a keyStoreUpdated invocation the engineInit method is invoked on each Factory object using the parameters originally used to initialize the class instance. This causes the PCKeyManagerFactory and PCTrustManagerFactory objects to re-initialize as described in (2). However, as the persistent handle stored in those instances has already been initialized, we simply update its internal reference to point to the updated X509KeyManager and X509TrustManager objects obtained from the reinitialized Factory objects, thus reflecting the update without requiring any reference changes inside the SSLContext.

It is necessary to synchronize access to instantiated PCTrustManager / PCKeyManager objects to ensure a change to the underlying manager object does not occur during some existing method invocation (e.g. one thread is setting up an SSL connection using the PCKeyManager while another is propagating a modification to the PCKeyStore on which it is based). Similarly the underlying Factory objects within PCTrustManagerFactory and PCKeyManagerFactory objects were protected to ensure conflicting access does not occur across threads (e.g. One thread invokes

`keyStoreUpdated` while another executes `engineInit`, possibly propagating some previous `PCKeyStore` modification).

## 3.5 HTTPS Protocol Support

Typically the developer, both at the server and client, will not want to work at the raw socket level in most cases. The final challenge was, therefore, to allow the developer to easily open HTTPS connections to servers using proxy certificates and handle client proxy certificates paths at the server. This section describes how this was accomplished.

### 3.5.1 Client HTTPS Support

In general, to establish an HTTPS connection the following is all that is necessary

```
URL url = new URL("https://some.address");
HttpsURLConnection urlCon = (HttpsURLConnection) url.openConnection();
```

However, to allow a named certificate to be used when setting up an HTTPS connection we would like to return a custom `HttpsURLConnection` from the call to `url.openConnection()`, which offers this ability, rather than Sun's default implementation. The relevant classes to achieve this are packaged in `com.hp.hpl.csf.security.https`. The `Handler` is a concrete implementation of `URLStreamHandler`, whose `openConnection(URL)` method simply returns a new `HttpsConnection`. `HttpsConnection` is a concrete implementation of `HttpsURLConnection` which also implements the `AliasNamer` interface to provide our extended functionality. The `AliasNamer` interface has a single method, `setAlias(String)`.

Much of the `HttpsURLConnection` implementation must be overridden. For example

- The abstract methods for accessing the local and server certificates as well as the cipher suite in use must be implemented.
- The connect method from `URLConnection` must be implemented to setup the SSL transport using the extended interface defined in our Provider that allows the naming of a local certificate.
- Implementing the connect method entails implementing the full HTTP/1.1 [15] compliant implementation, to parse headers and form requests.
- The default `set/getParameter` methods also had to be overridden.

### 3.5.2 Server HTTPS support

Jetty is a HTTP server and Java Servlet Container written entirely in Java which can easily be embedded in applications, or run as a stand alone application in its own right [12]. It is developed by the developers at Mort Bay consulting and made available under the Apache 2.0 license.

In Jetty, the server model for processing client HTTP requests is as follows:

[Listener(s)] → [HttpServer] → [HttpContext(s)] → [Handler(s)]

Listeners act as a source of requests for the HTTP server. `SocketListener` is the main implementation used which listens on standard TCP/IP ports for requests, although others also exist like that for SSL, non-blocking IO etc. The SSL listener interface is defined in the abstract `org.mortbay.http.JsseListener` class, concrete implementations of which can be configured to accept incoming requests to a Jetty server.

Our implementation overrides the `SSLServerSocketFactory createFactory` method inherited from `JsseListener`. The `createFactory` method is used to instantiate an `SSLServerSocketFactory` from which an `SSLServerSocket` will be created for accepting requests over. Within this method, a `PCKeyStore` is loaded, `PCTrustManagers` and `PCKeyManagers` obtained, a `PCSSLContext` created and finally a `PCSSLServerSocketFactory` obtained, all from our custom Provider implementations. The necessary configuration to make use of this is listed in the appendix, allowing the server accept requests that use proxy certificates for client authentication.

Servlets that are part of installed web applications can access the chain of proxy certificates that are associated with requests via the `javax.servlet.request.X509Certificate` request attribute.

# 4 Summary

This report outlines an approach to implementing proxy certificates described in [1] using the Java Cryptography API and related APIs. This was conducted as part of an evaluation of proxy certificates for deployment in a service-oriented architecture. Proxy certificates provide a convenient means to support single sign-on and delegated authority whilst utilizing existing infrastructure. The security provider described in the report should be useful for developers looking to add support for proxy certificates within existing applications.

# 5 Appendix A

This appendix provides a summary of the main Java classes and interfaces created to support the proxy certificates using the Java Cryptography API.

## 5.1 Package com.hp.hpl.csf.security

| Interface Summary | |
|---|---|
| `AliasNamer` | The `AliasNamer` interface provides a single method to allow application programmers to specify which certificate alias should be used from a `PCKeyStore` when establishing a secure connection over which an application level protocol will run. |
| `PCKeyStoreUpdateListener` | Listener interface to receive notifications about changes to the `PCKeyStore`. |

| Class Summary | |
|---|---|
| `PCertificate` | A `PCertificate` exposes accessor methods for the additional attributes present in an X.509 certificate containing a proxy certificate extension. |
| `PCGenerator` | A `PCGenerator` is used to create X509Certificate objects which may contain a proxy certificate extension. |
| `PCKeyManager` | A `PCKeyManager` encapsulates a `KeyManager` and allows a client alias to be specified to control which proxy certificate is used to establish an SSL connection. |
| `PCKeyManagerFactory` | `PCKeyManagerFactory` objects are used to instantiate `PCKeyManagers`, and dynamically ensure that their encapsulated `X509KeyManager` is kept up to date with changes to the `KeyStore`. |
| `PCKeyStore` | A `PCKeyStore` informs registered listeners of updates made to its contents. |
| `PCKeyStoreUpdateBroadcaster` | This singleton class broadcasts changes to the `PCKeyStore` to registered listeners. |
| `PCPathValidator` | A `PCPathValidator` is used to validate `CertPaths` of X509Certificates that might contain a chain of certificates using the proxy certificate extension. |
| `PCProvider` | A `PCProvider` implements a JCA Provider offering services for the creation, validation and management of X509Certificates which use the proxy certificate extension. |
| `PCSSLContext` | `PCSSLContext` implements an `SSLContext` which supports the creation of `SSLSocketFactory` objects that allow the explicit naming of a proxy certificate for use in the creation of an `SSLSocket`. |

| | |
|---|---|
| PCSSLSocketFactory | A PCSSLSocketFactory exposes an interface that allows the creation of sockets using a named certificate for the SSL handshake. |
| PCTrustManager | A PCTrustManager is an X509TrustManager that supports X509Certificate paths that may contain the proxy certificate extension. |
| PCTrustManagerFactory | PCTrustManagerFactory objects are used to instantiate PCTrustManagers, and dynamically ensure that their encapsulated X509TrustManager is kept current with updates to the PCKeyStore. |

## 5.2 Package com.hp.hpl.csf.security.https

For convenience additional classes were created to allow HTTPS connections using named proxy certificates to be easily created.

| Class Summary | |
|---|---|
| Handler | A custom handler for URL objects for the HTTPS scheme, which exposes an interface allowing the certificates to be specified. |
| HttpsConnection | An implementation of the abstract HttpsURLConnection class, which exposes the AliasNamer interface. This allows the developer to control which certificate held in the PCKeyStore is used to authenticate an SSL connection. |

## 5.3 Jetty Configuration

Below is an example of a Jetty configuration to support a secure socket handler which can process proxy certificates.

```
<?xml version="1.0"  encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure 1.1//EN"
"http://jetty.mortbay.org/configure_1_2.dtd">

<Configure class="org.mortbay.jetty.Server">
       <Call name="addListener">
              <Arg>
                     <New class="com.hp.hpl.csf.security.mortbay.PCJsseListener">
                            <Set name="Port">8443</Set>
                            <Set name="Keystore">c:\\certificates\\store.jks</Set>
                            <Set name="Password">test</Set>
                            <Set name="KeyPassword">test</Set>
                            <Set name="NeedClientAuth">true</Set>
                     </New>
              </Arg>
       </Call>
       <Call name="addWebApplication">
              <Arg>/</Arg>
              <Arg>C:\\eclipse\\workspace\\X509ProxyCert\\examples</Arg>
       </Call>
</Configure>
```

# References

[1] Von Welch, Ian Foster, Carl Kesselman, Olle Mulmo, Laura Pearlman, Steven Tuecke, Jarek Gawor, Sam Meder, and Frank Siebenlist. X.509 Proxy certificates for dynamic delegation. Proceedings of the 3rd Annual PKI R&D Workshop, 2004.

[2] R. Housley, W. Ford, W. Polk and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. IETF RFC 2459, January 1999.

[3] Steven Tuecke, Von Welch, Doug Engert, Laura Perlman, and Mary Thompson. Internet X.509 Public Key Infrastructure (PKI) Proxy certificate Profile. IETF RFC 3820, June 2004.

[4] Alan O. Freier, Philip Karlton and Paul C. Kocher. The SSL Protocol Version 3.0. Netscape Communications, November 18, 1996. http://wp.netscape.com/eng/ssl3/

[5] The Globus Alliance. Globus Toolkit. http://www.globus.org/

[6] Java CoG Kit. http://www-unix.globus.org/cog/distribution/1.1/API.html

[7] GridSite. http://www.gridsite.org/

[8] The OpenSSL Project. http://www.openssl.org/

[9] European DataGrid Java Security project. http://edg-wp2.web.cern.ch/edg-wp2/security/edg-java-security.html

[10] Sun Microsystems, Inc. How to Implement a Provider for the JavaTM Cryptography Architecture. http://java.sun.com/j2se/1.4.2/docs/guide/security/HowToImplAProvider.html

[11] Brian Maso. A New Era for Java Protocol Handlers. August 2000. http://java.sun.com/developer/onlineTraining/protocolhandlers/

[12] Jetty: Web Server and Servlet Container. http://jetty.mortbay.org/jetty/index.html

[13] Jessie - A free software implementation of the JSSE. http://www.nongnu.org/jessie/

[14] The Legion of the Bouncy Castle. http://www.bouncycastle.org/

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. IEFT RFC 2068, January 1997.