SignServer

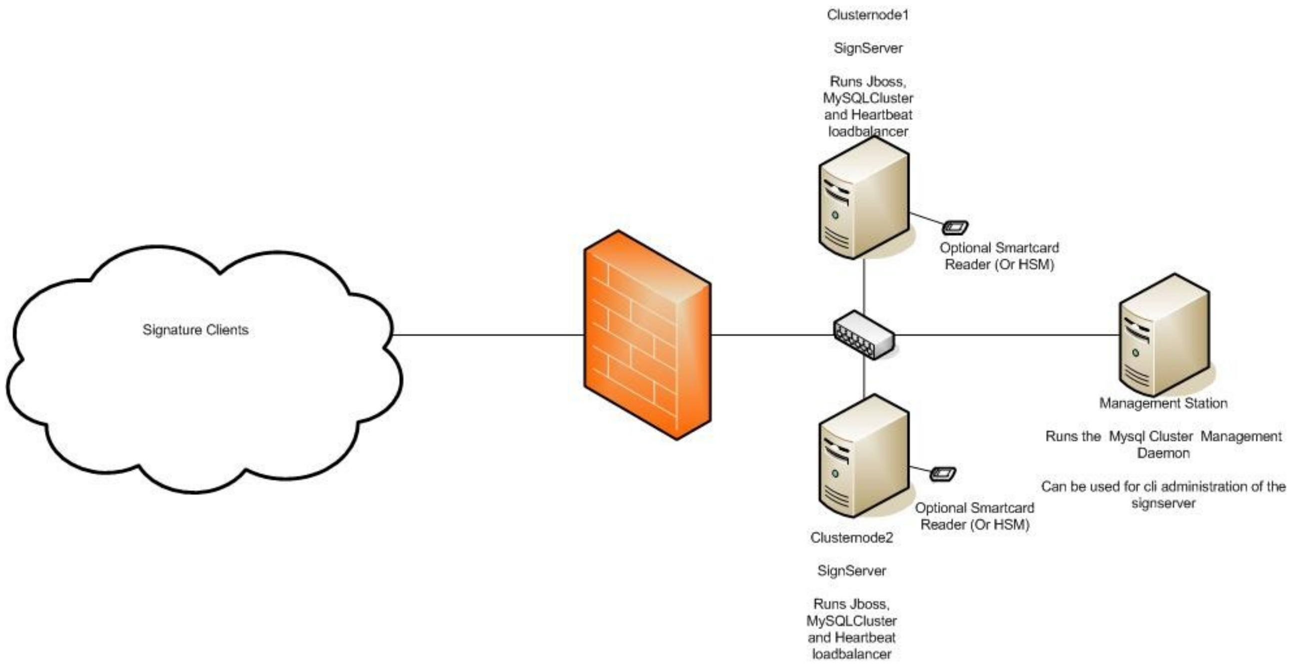
Manual

Ver: 2_0

07-08-10

1 Introduction/Scope

The SignServer is an application framework performing signatures for other applications.
It's intended to be used in environments where keys are supposed to be protected in hardware but there isn't possible to connect such hardware to existing enterprise applications. Another usage is to provide a simplified method to provide signatures in different application managed from one location in the company.
The SignServer have been designed for high-availability and can be clustered for maximum reliability.
The SignServer comes with a RFC 3161 compliant Time-Stamp signer serving requests through http or client-authenticated https. And a MRTD (Machine Readable Travel Document, i.e. electronic passport) signer.



Drawing 1: Overview of a possible set up of a highly available SignServer solution

1.1 Changes from Version 1

- signserver_server.property file have been removed and replaced with a global configuration store.
- It is now possible to dynamically and and remove available signers
- A new type of component, “Service” that is run on a timely basis, used to perform maintenance or report generation.
- Improved cluster deployment functionality.
- New CLI tools to batch configure the SignServer, and to backup a current configuration. This makes it possible to set-up a configuration in test environment, dump the configuration and configure it in production.

2 Document History

Version	*Date*	*Name*	*Comment*
0.1	2006-06-04	Philip Vendil	Initial version of this document.
1.0 RC1	2006-08-22	Philip Vendil	Prerelease version
2.0	2007-08-10	Philip Vendil	Update with new features for version 2

Table of Contents

1 Introduction/Scope....2
1.1Changes from Version 1....3
2 Document History....3
3 Quick-start of a Simple Time-stamp Server....5
3.1Required Software....5
3.2Installation Steps....5
4 Terms Used in This Document....7
5 Overall Architecture....8
6 Available Signers....9
6.1Time-stamp Signer....9
6.1.1Overview....9
6.1.2Available Properties....9
6.2MRTD Signer....10
6.2.1Overview....10
6.2.2Available Properties....10
7 Available SignTokens....11
7.1P12SignToken....11
7.1.1Overview....11
7.1.2Available Properties....11
7.2PrimeCardHSMSignToken....11
7.2.1Overview....11
7.2.2Available Properties....11
8 Setting Authorization Type....12
9 Disabling a Signer....12
10 Archiving Responses....12
11 The Global Configuration Store....12
12 Services....13
13 Building and Deploying the SignServer....13
14 Administrating the SignServer....14
15 Making the SignServer highly-available....17
15.1HTTP access requires a load balancer....17
15.2Setting up a MySQL Cluster....17
16 For Developers....18
16.1The ISigner Interface....18
16.2The ISignToken Interface....18
16.3The IService Interface....20
16.4Using the Global Configuration Store....20
17 Testing....22
17.1Automatic Junit Tests....22
17.2The Test Client....22
17.3Manual Tests....22
18 References....23

3 Quick-start of a Simple Time-stamp Server

This section will show how to set up a quick and simple standalone time-stamp server, accepting time-stamp requests over plain HTTP.

3.1 Required Software

- Java 1.5 (or 1.4) (http://java.sun.com)
- JBoss-4.0.5.GA (http://www.jboss.org)
- Ant version 1.7 (http://ant.apache.org)
- SignServer-2.0 (http://www.signserver.org)
- 1 web-server key-store in Java key-store format (JKS), (make sure that the server certificate have the right host name in it's CN)
- 1 Root certificate of the web-server in DER encoding
- 1 Time-stamp key-store in PKCS12 format

3.2 Installation Steps

1. First make sure that ant, Java and JBoss is installed properly.

2. Set the JAVA_HOME, JBOSS_HOME and SIGNSERVER_HOME environment variables.

3. Set the SIGNSERVER_NODEID environment variable, it should be a server unique string identifying the node in a cluster. (optional for one node installations).

4. Unzip the SignServer package and go to it's home directory.

5. If you are going to protect the HTTP communication with SSL, you need a JKS SSL server key store. Rename the web server key store to tomcat.jks at put it in a 'p12' subdirectory. Also place the web server root certificate in DER encoding in the same directory, call it rootcert.cer

6. Then copy the signserver_build.properties.sample file to signserver_buld.properties and edit the file. At least configure the httpsserver.password property. If you are not using https uncomment the row "`j2ee.web-nohttps=true`".

7. Do 'ant deploy' and then start JBoss (JBOSS_HOME\bin\run.sh) in another console.

8. Edit the signserver_cli.properties and set the host name.* properties.

9. Edit the qs_timestamp_configuration.properties file and set the default policy id, the path to the timestamp p12 file and its password.

10. Use the signserver cli to upload the configuration file. (if it's not executive use chmod +x bin/signserver.sh)

```
bin\signserver.sh setproperies qs_timestamp_configuration.properties
```

(In the path section, use '\\' for '\' in windows environment.)

Then run

```
bin\signserver.sh getconfig 1
```

Or you could use the signers name

```
bin\signserver.sh getconfig timestampSigner
```

And double-check the configuration. (Important, the properties are case sensitive).

Finally run

```
bin\signserver.sh reload 1
```

To activate the configuration.

11. Run the test-client to see that everything is up.

```
cd dist-client
java -jar timeStampClient.jar "http://localhost:8080/signserver/tsa?signerId=1"
```

`

The message "TimeStampRequest Validated" should appear once a second.

Also check JBOSS_HOME/server/default/log/server.log that successful messages appear.

4 Terms Used in This Document

Term	*Explanation*
Signer	A service performing signatures upon requests. This could be a ready made signer or a custom developed one.
Sign Token	A Sign Token is name for the entity containing the private key. Every Signer have a Sign Token that can be a PKCS12, Smart Card or HSM connection.
Service	A service is a task that is run on a timely basis, performing maintenance tasks like changing active key, or it could generate a report.
Worker	A common name for Signer and Service
Worker Configuration	Each Worker (Signer of Service) can be configured with properties specific for that worker. There are two sets of worker configuration one "Active" that is used by the signer and one "current" which is the one configured by the administrator. The current configuration isn't used in production until the administrator issued the reload command. This makes it possible for the administrator to configure multiple properties and double-check them before they are actually used.
Global Configuration Store	Is a dynamic store used to define available Signers and their Sign Tokens or Services. But other data that needs to be read globally could be set there as well. The global configuration properties are activated immediately. There are two different scopes for the store data, Global Scope and Node Scope.
Global Scope	Data stored in the global configuration that can be read by all nodes in the cluster.
Node Scope	Data that is node specific and can only be read within the same node.
Signer Id	Unique identifier of a signer, an integer larger than 0
Signer Name	A name used as a human readable synonym for a Signer Id

5 Overall Architecture

The SignServer is a framework designed to perform different kind of digital signatures for different applications.

To access the SignServer there are different options depending on the signer used. The time-stamp signer uses HTTP for remote access and the MRDT Signer uses RMI-SSL (requires commercial add-on from PrimeKey Solutions, www.primekey.se).

There are two main concepts to the SignServer and that is Signers (which performs the actual signature and are identified by a name or signerId) and SignToken which is an entity that keeps track of the signing keys (can be either software or hardware based).

The applications i administrated through a command-line interface, where the properties and access control can be configured.

The currently exists two authentication modes, one using client authenticated SSL where the serial number of each signer client have to be in the signers access list for the signature to be performed. The other is an unauthenticated mode where there isn't any requirements on who requests a signature.

One SignServer can have multiple signers for different purposes.

New to version two of the SignServer is the possibility to create services that does for example maintenance (or reports) on a timely basis.

Another new core feature is the possibility to dynamically add and remove signers to the system using a global configuration store. See the Global Configuration Store for more information.

6 Available Signers

There currently exists two types of signers. The first one is the time stamp signer generating RFC 3161 compliant timestamps using the Bouncycastle library. The other is a MRTD signer creating 'Machine Reader Travel Document' signatures using the RSA algorithm from pre-padded data.

6.1 Time-stamp Signer

The time-stamp signer have the class path: org.signserver.server.signers.MRTDSigner

6.1.1 Overview

The time stamp server generates time stamp tokens and have the support for the following options:

- Set of accepted policies
- Set of accepted algorithms
- Set of accepted extensions
- Accuracy microseconds
- Accuracy milliseconds
- Accuracy seconds
- Included certificate chain (currently doesn't include CRLs)
- Ordering
- TSA name

The time stamp signer currently don't support:

- CRL inclusion
- Signed attributes
- Unsigned attributes

Timestamps requests are served through a http service at the URL:
'http://<host name>/signserver/tsa?signerId=<signer Id>'
If no 'signerId' parameter is specified then will the id of 1 be used as default.

The time-stamp signer requires a time-stamp certificate with the extended key usage 'time-stamp' only.

6.1.2 Available Properties

The following properties can be configured with the signer:

> ***TIMESOURCE*** = property containing the class path to the ITimeSource implementation that should be used. (OPTIONAL, default LocalComputerTimeSource)
>
> ***ACCEPTEDALGORITHMS*** = A ';' separated string containing accepted algorithms, can be null if it shouldn't be used. (OPTIONAL, Strongly recommended)

Supported Algorithms are: *GOST3411, MD5, SHA1, SHA224, SHA256, SHA384, SHA512, RIPEMD128, RIPEMD160, RIPEMD256*

ACCEPTEDPOLICIES = A ';' separated string containing accepted policies, can be null if it shouldn't be used. (OPTIONAL, Recommended)

ACCEPTEDEXTENSIONS = A ';' separated string containing accepted extensions, can be null if it shouldn't be used. (OPTIONAL)

DEFAULTTSAPOLICYOID = The default policy ID of the time stamp authority (REQUIRED, if no policy OID is specified in the request then will this value be used.)

ACCURACYMICROS = Accuracy in micro seconds, Only decimal number format, only one of the accuracy properties should be set (OPTIONAL)

ACCURACYMILLIS = Accuracy in milliseconds, Only decimal number format, only one of the accuracy properties should be set (OPTIONAL)

ACCURACYSECONDS = Accuracy in seconds. Only decimal number format, only one of the accuracy properties should be set (OPTIONAL)

ORDERING = The ordering (OPTIONAL), default false.

TSA = General name of the Time Stamp Authority. (OPTIONAL)

6.2 MRTD Signer

The MRTD signer have the class path: org.signserver.server.signers.MRTDSigner

6.2.1 Overview

The MRTD Signer performs a RSA signing operation on incoming data. The data should already be padded. This signer i used to sign 'Machine Readable Travel Documents' i.e. electronic passports.

The signatures are requested through RMI-SSL and PrimeCard is required to build it. No load balancer is required to make it redundant, this is done with the client libs.

6.2.2 Available Properties

No configuration properties exists.

7 Available SignTokens

There exists two types of sign tokens, one storing the keys in software and one on SmartCards. See the developer section for information about developing support for other HSMs.

7.1 P12SignToken

The P12SignToken signer have the class path: org.signserver.server.signertokens.P12SignToken

7.1.1 Overview

A SignToken using a PKCS 12 key-store in the file-system. Can only contain one signing key.

In a clustered environment must the key store be at the same location at all nodes.

The P12SignToken, doesn't support the destroyKey() method

7.1.2 Available Properties

KEYSTOREPATH : The full path to the key-store to load. (required)

KEYSTOREPASSWORD : The password that locks the key-store. Used for automatic activation.

7.2 PrimeCardHSMSignToken

7.2.1 Overview

Using PrimeCardHSM it's possible to use a SmartCard to generate 2048-bit signatures. The SmartCard can perform about one signature a second.

PrimeCardHSM requires PCSCD software and SmartCard drivers. See separate documentation about installing PrimeCardHSM.

The PrimeCardHSMSignToken, doesn't support the destroyKey() method

7.2.2 Available Properties

defaultKey = Hash value of the signing key on the card. See PrimeCardHSM documentation for more information.(Required)

authCode = Authentication code for automatic activation (Optional).

8 Setting Authorization Type

By default is client-certificate authentication required for a signature request to be processed. This can be changed with the ***AUTHTYPE*** property.

AUTHTYPE = NOAUTH, sets the server to not require any authentication.

AUTHTYPE = CLIENTCERT (default) requires a certificate of all the clients. The certificates must be in the signers access control list and be trusted by the Java distribution, i.e imported in JAVA_HOME/jre/lib/security/cacerts.

9 Disabling a Signer

A signer can be disabled, which means that it won't perform any signature requests or be included in the health check.

To disable a signer set the property "DISABLED" to "TRUE"

10 Archiving Responses

If there is a need to save all generated responses, then set the property "ARCHIVE" to "TRUE" and all generated responses for that signer will be saved to database.

The archived responses can later be extracted from data base using the CLI interface. See the CLI section for more information.

11 The Global Configuration Store

The available signers and its sign tokens and services is configured in something called the global configuration store that is slightly different from a worker configuration.

Is is dynamically configured and activated immediately. I can contain any type of data (in string representation) and can be of two types, either with global scope or node scope. A Global scoped property can be read ny and is common to all nodes in the cluster while a Node scoped property in only used within a node and cannot be accessed by other nodes.

Is there is no activation of the global properties, database failure is handled differently. If a node looses connection to the database it put itself in a state called 'unsynchronised' a will continue operation without storing the data to database by using a cached global configuration. It is possible to later resynchronise a nodes global configuration data with a CLI command called 'resync'. But it is only possible to sync one of the nodes global configuration to the database.

12 Services

A service is a task that is run on a timely basis, performing maintenance tasks like changing active key, or it could generate a report.

Currently isn't the SignServer shipped with any services out of the box, but read the developer section about how to write custom services.

13 Building and Deploying the SignServer

How to set-up a highly available SignServer cluster is described in a separate document called SignServer_2_Installation_Guide.pdf that can be downloaded from http://www.signserver.org
The document describes how to set it up in a Cent OS 4.4 environment with two service nodes and one management node, but should be rather easy to adjust to other platforms.

14 Administrating the SignServer

The sign server is administrated using a CLI interface.

Every signer is identified by a id and optionally a name that can be used in all the CLI commands.

It is possible to do configuration of a signer or service while being in production. All configuration commands are cached until a reload command is issued and the configuration becomes active.

There is a special property file for the cli interface called signserver_cli.properties defining which nodes that exists in the cluster. The properties are:

> ***hostname.masternode*** = Should only contain one of the nodes, specified as the default master node. Used by operations dealing with the database and where not all nodes in the cluster needs to be contacted. It is possible to override this setting by using the -host <host name> parameter.
>
> ***hostname.allnodes*** = Should contain all the nodes in the cluster, separated by a ';'. Used by the commands getStatus, activateSignToken and deactivateSignToken.

The cli lies in bin\signserver.sh/cmd

Get Status Command:
Returns the status of the given signer, it says if it's sign token is active or not and the loaded 'active' configuration. It is possible to get a brief summary or a complete listing for one signer or all configured signers. If all signers are displayed will also all the global configuration parameters be displayed.

Get Configuration Command:
Returns the current worker or global configuration depending on options.

For worker configuration observe that this configuration might not have been activated yet, not until a 'reload' command is issued.

Set Property Command:
Sets a custom property used by the signer or signer token, see reference for the Signer and SignToken for available properties.

Set Properties Command
Command used to batch a set of properties, both for the global and worker configuration.
It can be used to configure a signer in a test environment, dump all the properties and upload it into production.

It reads all the configuration properties form a property file and depending on the contents of the key it sets the given property. All properties will be set according to the following defined rule set.

Rule	***Comment***
Properties starting with id<num>.	Will set the property to the value of the given id to the worker with the given id.
Properties starting with name<name>.	Will set the property to a worker with the given name. (If the name doesn't exists a unique id will be generated and assigned).
Property keys containing GENID<NUM>, example WORKERGENID1 or GLOB. WORKERGENID1	The SignServer will find a free unique id and assign substitute all GENID<num> with this id.
Properties starting with glob.	Will set a global property with global scope.
Properties starting with node.	Will set a global property with node scope.
Properties starting with -<other prefix><value>	Will remove the property, either worker or global.

See the file democonfiguration.properties for examples.

Remove Property Command:
Removes a configured property

Dump Properties
This tool will dump all configured properties for one or all workers in the system into a property file. If the configuration for one worker is dumped it can be used to transfer the configuration from one installation to another. If all configurations is dumped, it can be used as a backup tool.

Upload Certificate Command:
Used when updating the certificate used for signing, sign requests

Upload Certificate Chain Command:
Used when updating the TSA signer when using a Hard Signer Token. Use this one instead of 'Upload Certificate Command' in this case.

Generate Certificate Request Command
Used to generate a certificate request for a signer to be signed by a certificate authority. It takes distinguished name and signature algorithm as parameters and writes the request in PEM format to file.

Add Authorized Client Command:
Adds a client certificate to a signers list of acceptable clients using this signer. specify certificate serial number in hex and the Issuer DN of the client certificate.

Removes Authorized Certificate Command:
Removes added client certificates.

List Authorized Clients Commands:
Displays the current list of acceptable clients.

Activate Sign Token Command:
Used to activate hard sign tokens, authentication code is usually the PIN used to unlock the key store on the HSM. Not used if token is set to auto-activation.

Deactivate Sign Token Command:
Brings a Sign Token Off-line. Not used if token is set to auto-activation.

Archive Find from Archive Id Command:
Command used to extract archived data from database identified by the archive Id.

The Id depends on the signer, in case of the TSA is the TimeStampInfo serial number used.
The data is stored with the same file name as the archive id in the specified path.

Archive Find from Request IP Command:
Used to extract all archived data requested from a specified IP address.

All data is stored as separate files with the archive id as file name in the specified path.

Archive Find from Request Certificate Command:
Used to extract all archived data requested from a client by specified it's certificates serial number and issuer DN.

All data is stored as separate files with the archive id as file name in the specified path.

15 Making the SignServer highly-available

15.1 HTTP access requires a load balancer

HTTP based signers like the TSA can be clustered using a health check servlet returning the state of the signserver. The health check servlet can be configured in the file src/web/healthcheck/WEB-INF/web.xml. With the default settings will the servlet return the text 'ALLOK' when accessing the URL http://localhost:8080/signserver/healthcheck/signserverhealth. If something is wrong with the sign server will and error message be sent back instead.

The healthcheck servlet can also be used to monitor the signserver by creating a script that monitors the URL periodically for error messages.

Tip, heartbeat with ldirectord is a good solution for a load balancer and works well with the sign server.

15.2 Setting up a MySQL Cluster

The database backed of the SignServer can be made redundant using MySQL Cluster. Details on how to set-up the MySQL cluster can be found in the document SignServer_2_Installation_Guide.pdf that can be downloaded from http://www.signserver.org. More information about the MySQL Cluster can be found at http://www.mysql.com/products/database/cluster/

16 For Developers

To develop custom signers and sign tokens the following steps have to be done

16.1 The ISigner Interface

- Create a custom signer class implementing the org.signserver.server.signers.ISigner interface. There exists a BaseSigner that can be inherited taking care of some of the functionality. If the BaseSigner is inherited only one method signData() needs to be implemented.
- You can define your own properties that the signer can use for configuration.
- Make sure the custom class is available to the application server
- Redeploy the signer server and it the signer is ready to be configured and then used.
- Register the signer in the application by setting a property `WORKER<id>.CLASSPATH` with a global scope in the global configuration. (Also make sure to set it's sign tokens class-path, see next section).
- Reload the service with the CLI reload command.

16.2 The ISignToken Interface

- A custom sign token needs to implement the interface org.signserver.server.signtokens.ISignToken. See P12SignToken for an example implementation.
- You can define own properties for a sign token in the same way as for signers. The properties are sent to the sign token upon initialization.
- Make sure the custom class is available to the application server
- Redeploy the signer server.
- Register the signer token to a signer in the application by setting a property `WORKER<id>.SIGNTOKEN.CLASSPATH` with a global scope in the global configuration. (Also make sure to set it's sign tokens class-path, see next section).
- Reload the service with the CLI reload command.

The ISignToken interface have the following methods that needs to be implemented.

```
public interface ISignToken {
public static final int PURPOSE_SIGN = 1;
public static final int PURPOSE_DECRYPT = 2;
public static final int PROVIDERUSAGE_SIGN = 1;
public static final int PROVIDERUSAGE_DECRYPT = 2;
/**
* Method called after creation of instance.
*
*/
public abstract void init(Properties props);
/**
* Method that returns the current status of the sign token.
*
* Should return one of the SignerStatus..STATUS_.. values
*/
public abstract int getSignTokenStatus();
```

```
/**
* Method used to activate SignTokens when connected after being offline.
*
* @param authenticationcode used to unlock catoken,i.e PIN for smartcard
* HSMs
* @throws SignTokenOfflineException if SignToken is not available or connected.
* @throws SignTokenAuthenticationFailedException with error message if authentication to
* SignTokens fail.
*/
public abstract void activate(String authenticationcode) throws
SignTokenAuthenticationFailureException, SignTokenOfflineException;
/**
* Method used to deactivate Hard CA Tokens.
* Used to set a CAToken too offline status and to reset the HSMs
* authorization code.
*
* @return true if deactivation was successful.
*/
public abstract boolean deactivate();
/** Returns the private key (if possible) of token.
*
* @param purpose should one of the PURPOSE_... constants
* @throws SignTokenOfflineException if SignToken is not available or
* connected.
* @return PrivateKey object
*/
public abstract PrivateKey getPrivateKey(int purpose) throws SignTokenOfflineException;
/** Returns the public key (if possible) of token.
*
* @param purpose should one of the PURPOSE_... constants
* @throws SignTokenOfflineException if SignToken is not available or
* connected.
* @return PublicKey object
*/
public abstract PublicKey getPublicKey(int purpose) throws SignTokenOfflineException;
/** Returns the signature Provider that should be used to sign things with
* the PrivateKey object returned by this signingdevice implementation.
* @param providerUsage should be one if the ISignToken.PROVIDERUSAGE_ constants
* specifying the usage of the privatekey.
* @return String the name of the Provider
*/
public abstract String getProvider(int providerUsage);
/**
* Method returning the signertokens certificate if it's included in the token.
* This method should only be implemented by soft signtokens which have the certificate
* included in the keystore.
* All other signtokens should return 'null' and let the signer fetch the certificate from
* database.
*
*/
public abstract Certificate getCertificate(int purpose) throws SignTokenOfflineException;
/**
* Method returning the signertokens certificatechain if it's included in the token.
* This method should only be implemented by soft signtokens which have the certificates
* included in the keystore.
* All other signtokens should return 'null' and let the signer fetch the certificate from
* database.
*/
public abstract Collection getCertificateChain(int purpose) throws SignTokenOfflineException;
/**
* Method used to tell the signer to create a certificate request using its sign token.
*/
public ISignerCertReqData genCertificateRequest(ISignerCertReqInfo info) throws
SignTokenOfflineException;
/**
* Method used to remove a key in the signer that shouldn't be used any more
* @param purpose on of ISignToken.PURPOSE_ constants
* @return true if removal was successfull.
*/
public boolean destroyKey(int purpose);
```

```
}
```

16.3 The IService Interface

There are two kinds of services, singleton or non-singleton. A singleton service is only run at one of the nodes at the time while non-singleton services are run at all nodes simultaneously. If a singleton service fails to run on one of the nodes will one of the other nodes take over the service automatically.

If a service should be singleton or not is determined by a standard property ***SINGLETON*** defined in the ServiceConfig class.

Other basic properties used to configure all services are:
ACTIVE when set to `"TRUE"` means that the service is active and should be run.
INTERVAL defining the interval in seconds of how often the service should be run.

To create a customized service perform the following steps:

- Create a custom service class implementing the org.signserver.server.service.IService interface. There exists a BaseService that can be inherited taking care of most of the basic functionality. If the BaseService is inherited only one method work() needs to be implemented.
- You can define your own properties that the service can use for configuration.
- Make sure the custom class is available to the application server
- Redeploy the signer server and it the signer is ready to be configured and then used.
- Register the service in the application by setting a property `WORKER<id>.CLASSPATH` with a global scope in the global configuration.
- Reload the service with the CLI reload command.

The work method that needs to be implemented is described here:

```
/**
* Method that should do the actual work and should
* be implemented by all services. The method is run
* at a periodical interval defined in getNextInterval.
*
* @throws ServiceExecutionFailedException if execution of a service failed
*/
public void work() throws ServiceExecutionFailedException;
```

16.4 Using the Global Configuration Store

The global configuration store is a memory bank that signers and services can use to store data used in ongoing operations. The data can be either node (i.e. only read by the current node) or global scoped.

To access the global configuration store use the `getGlobalConfigurationSession()` method from the BaseWorker (inherited by both the BaseSigner and BaseService). The returned GlobalConfigurationSession have the following methods that can be used (the other should be avoided)

```
/**
* Method setting a global configuration property. For node. prefix will the
node id be appended.
* @param scope, one of the GlobalConfiguration.SCOPE_ constants
* @param key of the property should not have any scope prefix, never null
* @param value the value, never null.
*/
public void setProperty( java.lang.String scope,java.lang.String
key,java.lang.String value ) ;
/**
* Method used to remove a property from the global configuration.
* @param scope, one of the GlobalConfiguration.SCOPE_ constants
* @param key of the property should start with either glob. or node., never
null
* @return true if removal was successful, othervise false.
*/
public boolean removeProperty( java.lang.String scope,java.lang.String key )
;
/**
* Method that returns all the global properties with Global Scope and Node
scopes properties for this node.
* @return A GlobalConfiguration Object, nevel null
*/
public org.signserver.common.GlobalConfiguration getGlobalConfiguration( ) ;
```

The getGlobalConfiguration returns a GlobalConfiguration and have a method `String getProperty(String scope, String property)` that can be used.

The value of the property can be user-defined as long as it is guaranteed to be unique over the entire application.

Reserved values are all property keys starting with "`WORKER`".

17 Testing

There exists some test scripts used to test that the SignServer functions correctly. They are described here.

17.1 Automatic Junit Tests

Automatic Junit tests lies in the directory '`src/tests`'

To run the test suite do the following:

- Set the environment variable SIGNSERVER_HOME
- Make sure the sign server is deployed and JBoss is running
- do 'ant test:run'

17.2 The Test Client

There exists a test client built with the main distribution.

It only works without client authentication requirements and through HTTP.

To run the client do

```
ant
cd dist-client
java -jar timeStampClient.jar "http://<hostname>:8080/signserver/tsa?signerId=1"
```

It will continuously make one request per second.

17.3 Manual Tests

The time stamp signer have been tested with the OpenTSA client with both HTTP and HTTPS.

18 References

Java 1.6 to 1.4 (http://java.sun.com)
JBoss-4.0.5.GA (http://www.jboss.org)
Ant version 1.7(http://ant.apache.org)
Bouncycastle (http://www.bouncycastle.org)
RFC3161, Time-Stamp Protocol (TSP) (http://www.ietf.org)