SHS Version 1.2.01 Application Interfaces

Verva - Swedish Administrative Development Agency

Editors:

Kurt Helenelund, Stephan Urdell, Bo Sehlberg, Anders Bremsjö, Anders Lindgren, Jan Lundh, Christer Marklund,

Contents

1 Introduction

This document is an introduction to the API specifications for the SHS version 1.2. In addition, details of the C API and the standalone programs (shssend/shsfetch) is included. Details about Java API and the SHS internal web service interface is found in separate documents.

1.1 Audience

This document is intended for developers and system integrators that need to understand how SHS API:s can be used.

1.2 References

[DTD] SHS 1.2.01 DTD Description

[Protocols] SHS 1.2.01 Protocol

[JAVA] SHS API 1.2.01 Javadoc documentation, see www.verva.se/shs/

[IWSI] SHS Internal Web Service Interface 1.0.01

1.3 Document history

Version	Date	Change	By	Approved
0.9.2	1999-04-08	Translated to English. Added/changed program API arguments. Added message list DTD.	Bo Sehlberg	
0.9.3	1999-04-15	After comments from David Kågedal. Added message interaction description.	Bo Sehlberg	
0.9.4	1999-04-27	Some last minute corrections	Bo Sehlberg	
0.9.5	1999-05-21	Minor corrections	Bo Sehlberg	
0.9.9	2000-03-08	Updated after comments	Bo Sehlberg	
1.0	2000-03-28	Updated after review	Bo Sehlberg	Yvonne Palminger
1.0.1	2000-11-21	Updated for SHS version 1.1	Bo Sehlberg	
1.0.2	2001-03-14	Minor corrections after review	Bo Sehlberg	
1.0.3	2001-04-27	Corrections after second review	Bo Sehlberg	
1.0.4	2001-04-30	Some more minor corrections	Bo Sehlberg	
1.0.5	2001-05-11	Adjusted command line args. added time format	Bo Sehlberg	
1.0.6	2001-05-23	Minor layout corrections	Bo Sehlberg	
1.0.7	2001-06-11	Added description for “-d <dir>	Bo Sehlberg	
1.1	2001-09-16	Final version minor corrections	Kurt Helenelund	Christer Marklund
1.1.1	2003-03-27	First draft of 1.2 documentation • Moved protocol descriptions to protocol document • Added an overview • Added placeholders for WS and Java descriptions	Anders Lindgren	

1.1.2	2003-05-27	Second draft for review meeting 2003-06-03 • Replaced placeholders for Java API and web services with references to other documents	Anders Lindgren	
		replace “document” with “message” replace “product” with “product type”	Anders Lindgren	
1.2	2003-10-09	Inclusion of Internal WS Final version 1.2	Anders Lindgren	Jan Lundh
1.2.01-A	2004-05-19	First draft of updated IWSI specifications • Changes to Web Service interface – new verion 1.0.01	Anders Lindgren	
1.2.01-B	2004-05-26	Second draft (B) of updated IWSI. • Comments from SHS ÄR 040526	Anders Lindgren	
1.2.01-C	2004-06-04	Third draft (C) – split of document into overview (this document) and IWSI documemtation	Anders Lindgren	
1.2.01-D	2005-02-01	Fourth draft • add Java API overview	Anders Lindgren	
1.2.01	2007-04-05	Final sub version	Anders Lindgren	Christer Marklund Jan Lundh

1.4 Document Conventions

In this document the following conventions are used:

Times New Roman	Normal text
[REFERENCE]	References are normal text (UPPER CASE) enclosed in square brackets.
`Courier 10pt`	Technical details such as command examples and definitions of environment variables

2 Application Interface overview

SHS defines various API levels. These API's represent abstraction levels with different intended usage.

The lower level API allows the user to manage details of SHS message creations and is primarily intended for those who intend to build an agent to other messaging systems such as IBM's MQ. On the other hand the highest level API will allow simple integration with end systems.

2.1 Standalone programs

For implementation in a scripting environment there are two standalone programs available.

- shssend – assembles and optionally encrypt and sign a SHS message and submits the data to a SHS node
- shsfetch – list, fetch, optionally decrypt, and disassemble SHS messages

2.2 C API

This is an API where all functionality in the SHS protocol level functionality is available to the programmer.

This assumes that the programmer builds all components of the message correctly but on the other hand gives the freedom to manipulate and construct data.

This API is primarily intended for full functional agents that ISV's and others may build for various applications such as message and queuing systems, ERP systems and so forth.

2.3 Java API

This is an API where all protocol level functionality in the SHS interface is available to the programmer.

This API defines a standardised interface enabling client applications written in Java to communicate with an SHS node. In order to support a Java client applications there are two required components:

- The standardised interface definitions available from Statskontoret
- An implementation of the classes defined in the interface. This package is available from SHS Java API compliant SHS vendors.

This API supports the basic objects for creating, sending, listing and fetching SHS messages. In addition a selection of support classes are available in the interface.

Java code samples is found in chapter 6 (examples). For a detailed description of the Java API, please see the Java API documentation [JAVA].

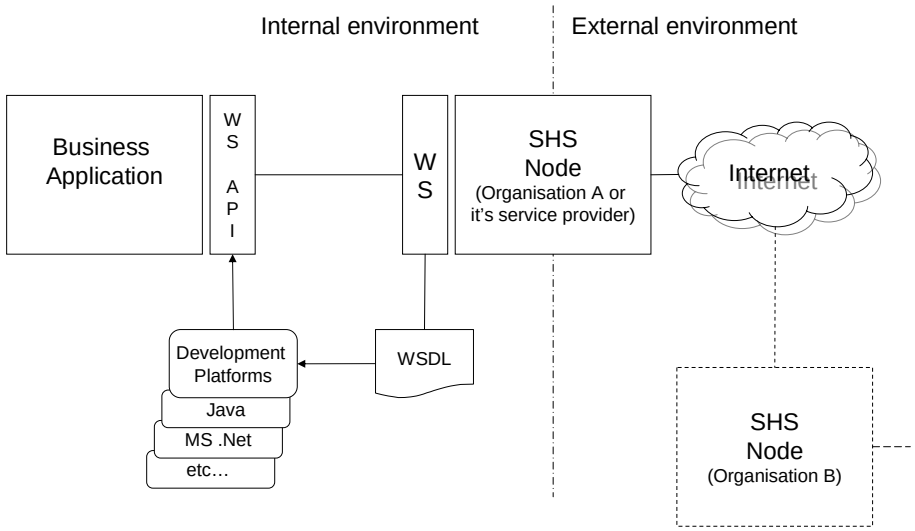
2.4 Internal Web Services Interface

This interface offers access for business systems to SHS functionality using W3C specifications for Web Services.

The Internal Web Services Interface version 1.0 is compatible with SHS 1.2. This interface provides a complement to the existing interfaces that:

- require a minimum of installed SHS specific components or libraries
- benefit from the tools/development platforms that are developed for the web services technology
- is platform independent

This interface is named SHS Internal Web Services Interface since it provides an interface for business applications to access SHS services. These services include (but is not limited to) reliable and secure transport of information and audit logging. SHS provides the connection to other organisations or applications, defined as “external environment” in the figure below.



SHS Internal Web Services Interface version 1.0 is restricted to listing, fetching and ending of messages with up to 1 MB[1] data parts. Signing and encryption is not supported in version 1.0. The interface is based on WSDL version 1.1 and SOAP version 1.1 and is further described in [IWSI]

3 Programming guidelines

SHS provides an infrastructure och middleware services to business applications for secure and reliable transfer of information. For system architects and programmers the following are the most important features of SHS:

- Delivery using a store and forward mechanism (aynchronous mode)
- Routing of messages based on a business oriented addressing concept
- Error handling and activity logging
- Reliable transfer of and tracing of messages

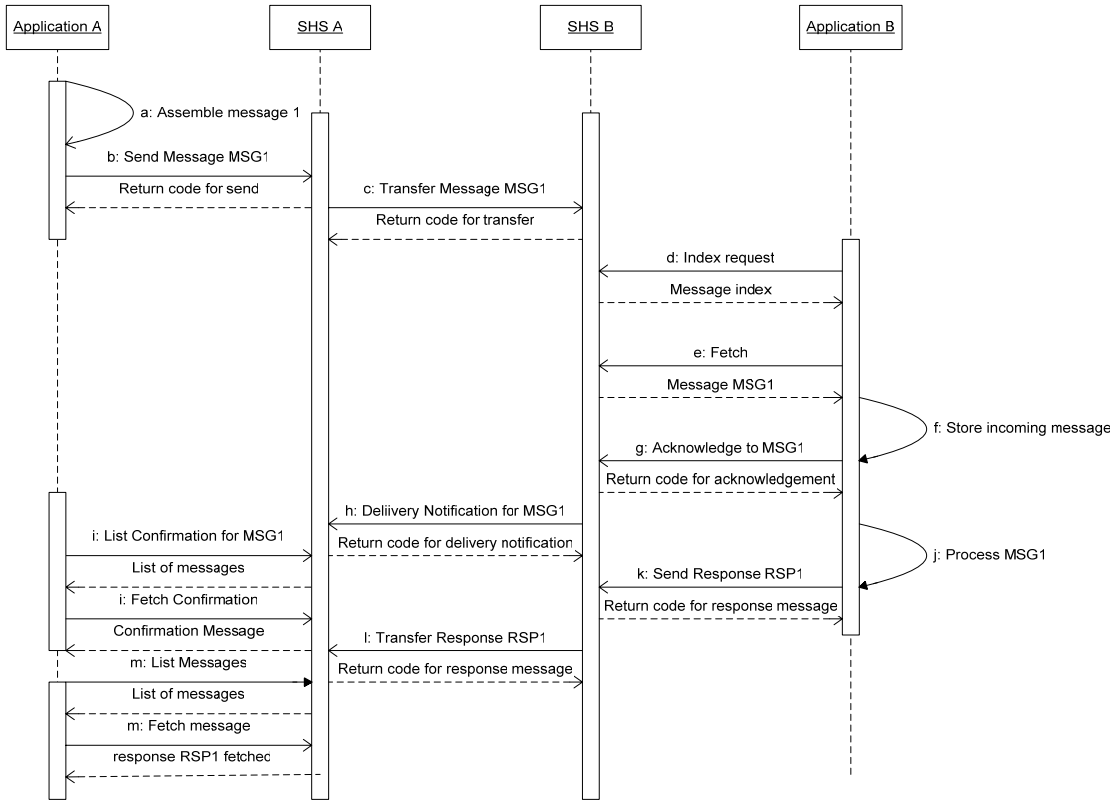
SHS supports two mayor communication modes – synchronous and asynchronous. In asynchronus mode more than one application may receive the message.

When an application initiates a synchronous transfer the call returns either the results from the called business application or an error status. SHS routes messages and logs the traffic.

3.1 Asynchrounous communication

Under numerous cirumstanses it is not possible for an application to rely on immediate response from a service. This may be caused by different reasons like system availability, processing time and information size. In SHS this is addressed with support for asynchronous communication between systems. The sending application submits a message and does not have to wait for the responder to answer. Note that the applications A and B in some cases are not active at same time.

[1] When encoding of the message in base64, the actual data transferred is 1.33 times larger than the payload



In the scenario the business application A submits a document for business application B. Comments to the sequences above (operations performed internally in an SHS node are written in *italic*):

A. Application A uses the API routines to create (assembe) an SHS message. One or more dataparts are added as defined by the product type and may optionally be signed or encrypted.

B. Application A calls a asynchronous send routine. Note that in the Internal Web Services Interface message creation and submit is performed by a single service.

C. The SHS node(s) logs the message, routes it to the destination based on the addressing information. Trace information is logged by each SHS node that handles the message. Finally the message reach its destination and is stored in business application B's outbox.

D. Business application B does a periodical search for new messages with a listing routine, which returns a message list, optionally filtered and sorted, of available messages.

E. This list may then be used to fetch the available messages.

F. When the business application has received the message successfully it stores the message for local processing in this case. May also process the message directly.

G. Business application B acknowledges the receipt to it´s SHS node.

H. If confirmation is requested the SHS node genererats a confirmation message that is routed back to SHS node.

I. Business application A may later list and fetch the confirmation message

J. Later the business application B may perform the actual processing which may result in a response message (RSP)

K. Application B assembles and sends (in IWSI again it's a single function) the response to SHS

L. The response message is logged and routed back to the application A's in-box

M. Business application now may list and fetch the response

3.2 Function and method summary

The following table is a summary of functions and methods that are used in the different API and interfaces:

Function	**C-API**	**Java**	**IWSI**	**Send/fetch**
Communication	SHS_Init, SHS_Disconnect	Class: ShsFactory Methods: createSendService createFetchService	N/A	N/A
Message assembling/ disassembling	SHS_Newlabel, SHS_Freelabel, SHS_addChild, SHS_addAttrib, SHS_findChild, SHS_nextChild, SHS_findAttrib	Class: Label Methods: getContent, get*DifferentInfo*, set*DifferentInfo* Class: LabelContent Methods: addDataPartInfo, getDataPartInfo	N/A[2]	N/A
Send functions	SHS_writeLabel, SHS_writeData	Class: ShsSendService Methods: createAsyncMessageOutputStream, createSyncMessageOutputStream Class: AsyncMessageOutputStream Methods: putNextDataPartEntry, write, close Class: SyncMessageOutputStream Methods: writeLabel, putNextDataPartEntry, write, close, getResponse	SHSSendSync, SHSSendAsync	shssend

[2] Included in send/fetch functions

Function	C-API	Java	IWSI	Send/fetch
List	SHS_readIndex	Class: ShsFetchService Method: getMessageInfoList Class: MessageInfo Method: get*DifferentInfo*	SHSListMessages	shsfetch
Fetching of messages from SHS	SHS_readLabel, SHS_readData	Class: ShsFetchService Method: getMessageInputStream, acknowledgeMessage Class: MessageInputStream Method: getLabel, getNextDataPartEntry, read, close	SHSFetchMessage, SHSAcknowledge	shsfetch (-i)
Encryption/ decryption	SHS_encrypt, SHS_decrypt	Class: EncryptionConfig Method: *new(Certificate),* setAlgorithm Class: MessageInputStream Method: setDecryptionKey	Not supported	Parameters to shssend and shsfetch
Sign/verify functions	SHS_sign, SHS_verify	Class: SignaturerConfig Method: *new(Certificate, PrivateKey),* setDigestAlgorithm Class: MessageInputStream Method: getSignatureCertificate	Not supported	Parameters to shssend and shsfetch
Other (support)	SHSstrerror	*Help and utility classes*		

3.3 Status and error handling

The follwoing is major types of error status handling

- Transport related status and error (http return codes)
- Confirmations
- Error messages
- SHS error codes returned by synchrounous calls

A description of the basic error codes is available in [Protocols] chapter 8.

A business application should always honour error and confirmation messages provided by the SHS API and interfaces. For an exact description of the interfaces please see the implmentor specific documentation.

4 Standalone programs

The standalone API consists of two standalone programs to be used to simplify the integration of a business system to SHS.

One program (shssend) is used for sending messages to other system through the SHS. The other program (shsfetch) is used for retrieval of messages from the SHS. These programs also require that the product type specification files are locally available for the program.

These programs are controlled by command parameters and environment variables. The parameters are described for each program. The environment variables are:

```
SHS-ADDRESS=<specifies own urn>
SHS-PRODUCT TYPE=< product type description file name or path to product type description file >
SHS-CERT=<file name of certificate file> or empty line if card-based certificate
SHS-KEY=<file name of private key file>
```

4.1 shssend

The shssend may be used interactively or from a script file. The following format may be used:

```
shssend  ([-O <orig>] –f <from>)|-O <orig> [-t <addr>] [-p <prod>] [-E <end-recipient>] [ -r <corrid> ][-e  <cert>][ -c ][-C <content id>] [-T] [-q <sequence type>] [-m <name>=<value>] [-s <subject>] ( ([-d <dir>]  <filename> ...)| -d <dir> )
```

The arguments for shssend have the following meaning:

`-f <addr>` Use `<addr>` as the "from" address.

`-O <orig>` Use `<orig>` to specify originator. This is used when the <addr> is unknown and may be a person's social security number, e-mail address, etc.

`-E <end-recipient>` This is used to further identify the recipient and may be a person's social security number, e-mail address, etc.

`-t <addr>` Use `<addr>` as the "to" address, this argument is optional.

`-p <prod>` Use `<prod>` to determine the product type and packaging characteristics. The argument <*prod*> should be a file name of the XML product (without the suffix .xml) description file, either a full path name, or a simple file name that will be searched for in a standard location. Product may also be one of the pseudo product types (error, confirm, agreement)

`-r <corr.id>` Correlation ID of a corresponding request, this argument is optional.

`-e <certificate>` Receiver certificate file name, when using encrypted message.

`-c` Specifies that the SHS message is a compound message, this argument is optional.

`-C <content id>` Specifies the content ID. If omitted the content id will be generated internally.

`-d <dir>` Directory path for business documents to be sent. If used without a filename list, all files in that directory will be sent.

<*filename*> ... Names of files to include in the SHS message, in the order corresponding to the 'product type' description.

`-T` Indicates to SHS that the message is a production test message.

`-q <sequence-type> where sequence-type::= (event|request|reply|adm)` Specifies the type of message.

`-m` <*name*>=<*value*>, used to include meta data in the label.

`-s` <*subject*>, used to include a subject in the label.

The execution will produce the following output:

Return:

Error: Exit code != 0

Success: Exit code = 0

Standard output:

Transaction ID

Standard error:

Error message

4.2 shsfetch

The shsfetch may be used interactively or from a script file. The following format may be used:

```
shsfetch -p <prod> [-t <addr> [-E <end-recipient]] -i <tx.id> [-r
<corrid>] [-T] [-q <sequence type>] [-m <meta data>] -d <dir> [ -s ] [
-o ]
```

The following input parameters are used by shsfetch:

`-t <addr>` Use `<addr>` to specify own address, this argument is optional.

`-E <end-recipient>` This is used to further identify the recipient and may be a person's social security number, e-mail address, etc.

`-p <prod>` Product type to fetch.

`-r <corr.id>` Correlation id of message to fetch.

`-i <tx.id>` Transaction id of message to fetch.

`-d <dir>` Directory path to put the resulting business documents.

`-s` Retrieve single files. If omitted all available files for specified product will be fetched.

`-o` Use original file names. If omitted, the transaction id (UUID) will be used in file names, see below.

`-T` Indicates to SHS that only test message should be fetched.

`-q <sequence-type> where sequence-type::= (event|request|reply|adm)` Specifies the type of message to fetch.

`-m <name>=<value>,` used to search for specific meta data in the label. Note: Only exact matching is supported (no wild card).

The program retrieves the SHS message and extracts the included files and saves the files in the specified directory. The files are stored in two possible manners:

- using the transaction UUID as the main filename
  ```
  <UUID>-label
  <UUID>-1
  <UUID>-2
        :
  <UUID>-n
  ```
- using the file name(s) provided in the message. In case duplicate file names are found an extension ~<number> will be added to the new filename to make it unique.

The execution will produce the following output:

Return:

Error: Exit code != 0

Success: Exit code = 0

Standard output:

Result message

Standard error:

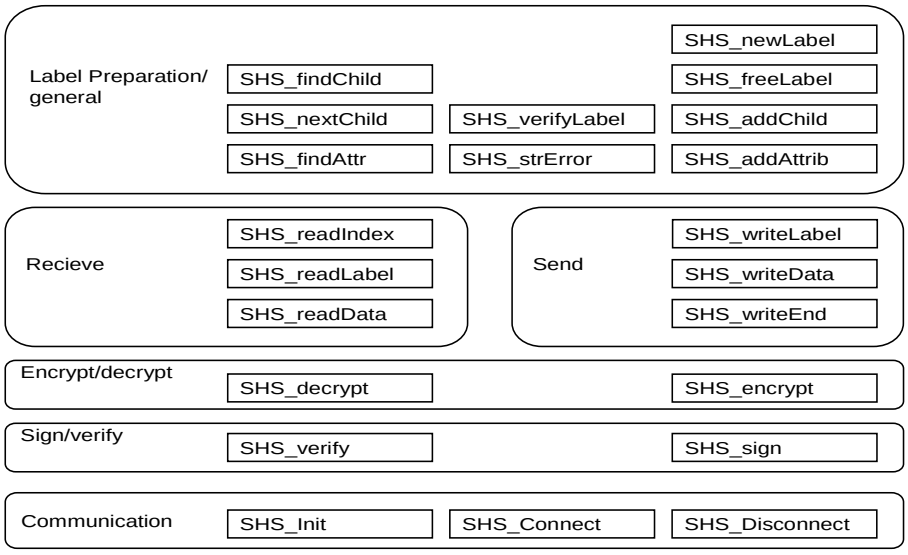
Error message

5 C API

The function API consists of a number of functions to allow an application to be integrated with the SHS. The API provides support for sending and retrieving SHS messages to SHS.

In order to allow both SSL session reuse and the possibility to send multiple requests over an HTTP connection, the connection phase is split into multiple steps.

The first step initialises SSL [SHS_init] secondly the connection is established with the server [SHS_connect]. A number of requests may be performed in sequence using the same connection. When the server disconnects due to timeout or other failure, the [SHS_write*] or [SHS_read*] calls will fail. To re-establish the connection a new call to the [SHS_connect] needs to be performed, unless the auto reconnect flag is set in the [SHS_connect] call.



The functions can be divided in six groups:

- Communication functions – the SSL handling
- Label preparation functions – message assembling/disassembling
- Send functions – posting of messages to SHS
- Read functions – fetching of messages from SHS
- Encryption/decryption functions
- Sign/verify functions

5.1 Communication functions

The communication functions are used to establish and terminate a session between the application and the SHS.

5.1.1 SHS_Init

Initialises the SSL parameters for the communication. The function retrieves required authentication information, related to provided arguments.

This function is called once during start-up/initialisation and is used in following connections.

Arguments:

pointer to authentication data returned by the function

private key path and file name

certificate path and file name

Return:

Success = 0

Error code <0

5.1.2 SHS_connect

This establishes the communication session to the SHS specified as an argument. A stream is returned for, sending and receiving SHS messages.

Arguments:

pointer to authentication data (NULL if no security is to be used)

pointer to connection data structure (including stream, status)

auto reconnect flag

Return:

Success = 0

Error code <0

5.1.3 SHS_disconnect

Terminates a communication session to an SHS. Used to terminate all communication.

Arguments:

pointer to connection data stream

Return:

None

5.2 Label preparation functions

The label preparation functions provide a flexible function API for the administration of the SHS label.

5.2.1 SHS_newLabel

Creates a top label element for an XML-document. This top element is a placeholder for the document root.

Arguments:

None

Return:

pointer to the allocated top element

NULL in case of error.

5.2.2 SHS_freeLabel

Deallocates the whole label. This means all elements and their children including the top node.

Arguments:

pointer to the label root element

Return:

None

5.2.3 SHS-addChild

Add a child element to a current element (the element that will contain the child). The new child will be given a name and a type. The elements in the XML document will be created in the same order as added. No control for correctness of the passed name is done (=no validation against the specified DTD is performed in the call). It is the user's responsibility to make sure it corresponds to the specified DTD. The type argument specifies whether the element is a processing instruction element, a document type element, a comment element or named element.

Arguments:

pointer to the current element

name of the child element

type of the element

Return:

pointer to the allocated child element

NULL in case of error.

5.2.4 SHS_addAttrib

Add a named attribute with specified value, to a previously allocated XML element (argument current element). No control for correctness of the passed name is done (=no validation against the specified DTD is performed in the call). It is the user's responsibility to make sure it corresponds to the specified DTD.

Arguments:

pointer to the current element

name of the attribute

Return:

pointer to the allocated attribute

NULL in case of error.

5.2.5 SHS_verifyLabel

Verifies a built label that it follows agreed grammar.

Arguments:

pointer to the root element

Return:

0 if the grammar is correct

< 0 if an error has occurred.

A textual description of the grammar error can be fetched with SHS_strerror.

5.2.6 SHS_strerror

Converts an error number returned by the SHS_* functions to a descriptive text. The text string is available until the next call to SHS_strerror.

Arguments:

error number

Return:

string describing the error

5.2.7 SHS_findChild

Function to find child element with a given name. Matching is case insensitive.

Arguments:

pointer to the current element

name of the element to search for

Return:

pointer to the found child element

NULL if not found.

5.2.8 SHS_nextChild

Optional function to find next element with a given name, starting from the current element (as found through the SHS_findChild). The current child is managed by the XML tree. Matching is case insensitive. Current element is updated to the next current element by SHS_nextChild.

Arguments:

pointer to the current element

name of the element

Return:

pointer to the found element

NULL if not found.

5.2.9 SHS_findAttrib

Optional function to find an attribute with a given name in an element. Matching is case insensitive.

Arguments:

pointer to the current element

name of the attribute

Return:

pointer to the found attribute

NULL if not found.

5.3 Send functions

5.3.1 SHS_writeLabel

Writes an XML document from the passed node hierarchy to an out stream. No validation is done of the written document.

Arguments:

pointer to the label root element

output stream

Return:

Number of bytes written to the out stream, in case of error the error code < 0, is returned.

More detailed error description can be fetched from SHS_strerror.

5.3.2 SHS_writeData

This function is used to send the message data to the opened stream after the [SHS_writeLabel].

Arguments:

output stream

input data

MIME content type

MIME transfer encoding

Return:

Success = 0

Error code < 0

5.3.3 SHS_writeEnd

This function is called to terminate the transfer and complete the MIME formatting.

Arguments:

stream

Return:

Success = 0

Error code < 0

5.4 Read functions

5.4.1 SHS_readIndex

Retrieves a list of files available to retrieve according to specified filter criteria.

Arguments:

input stream

filter criteria

data buffer size

pointer to data buffer

Return:

Success = 0

Error code < 0

5.4.2 SHS_readLabel

Reads an XML document from the passed in stream and builds the node hierarchy. No validation is done of the parsed message. The label data is returned into the same type of structure that is being used for [SHS_writeLabel]. This means changes and additions are easily done to an existing label.

Arguments:

input stream

Return:

pointer to the root element of the node hierarchy, NULL in case of error.

5.4.3 SHS_readData

This function is used to retrieve the message data from the opened stream after the [SHS_readLabel]. The function is called multiple times (for each data part in the SHS message) until it returns an end of file code.

Arguments:

input stream

input buffer size

pointer to data buffer

pointer to MIME content type

pointer to received size

Return:

End of file = 0

End of part in the SHS message = 1

Error code < 0

5.5 Encrypt/decrypt functions

5.5.1 SHS_encrypt

Reads a message from the input stream and sends the encrypted version to the output stream, using the specified cipher and recipient certificate.

Arguments:

input stream message

output stream encrypted message

cipher name

X.509v3 certificate file name of recipient

Return:

Success = 0

Error code < 0 if failed

5.5.2 SHS_decrypt

Reads an encrypted message from the input stream and sends the decrypted version to the output stream, using the specified certificate, private key and CA certificate.

Arguments:

input stream encrypted message

output stream decrypted message

X.509v3 certificate filename

own private key

CA certificate filename (X.509v3 cert)

Return:

Success = 0

Error code < 0 if failed

5.6 Sign/verify functions

5.6.1 SHS_sign

Reads a message from the input stream and sends the calculated signature to the output stream, using the specified digest algorithm, owner certificate and private key.

Arguments:

input stream message

output stream signature

digest algorithm

own private X.509v3 certificate filename

private key filename

Return:

Success = 0

Error code < 0 if failed

5.6.2 SHS_verify

Reads a message and corresponding signature from the input stream and verifies the signature, using the CA certificate.

Arguments:

input stream message

input stream signature

CA X.509v3 certificate filename

Return:

Success = 0

Error code < 0 if failed

6 Samples

6.1 Java code samples

In this chapter three simple Java examples show how to send, list and fetch SHS messages.

The samples uses a dummy SHS node with the following configuration

- The receive service have the address (url) *https://shs.node.se:11292/rs*
- The delivery service url is *https://shs.node.se:11292/ds*

Error handling is intetionaly left out in the samples. Applications used in production must of course handle exceptions.

6.1.1 Send an asynchronous SHS message

```
import java.io.IOException;
import java.net.MalformedURLException;
import se.statskontoret.shsapi.*;

. . .

    public void sendAsyncShsMessage()
        throws MalformedURLException, ShsConnectionException, IOException
    {
        // sender actor ID
        String from = "5544331234";

         // receiver actor ID
        String to = "5566771234";

        // product type ID
        String productType = "9ddfd1cd-11ee-faa6-3224-f192e50614b0";

        // SHS receiving URL
        String sendServiceUrl = "https://shs.node.se:11292/rs";

        // data part content
        byte[] testData = "<test>This is test data</test>\r\n".getBytes();

        // create service object for communicating with SHS
        ShsSendService sendService =
            ShsFactory.createSendService(sendServiceUrl);

        // create stream for sending an SHS message
        AsyncMessageOutputStream out =
            sendService.createAsyncMessageOutputStream();

        // create and write label
        Label label = new Label(from, to, productType);
        String txId = label.getTxId();
        LabelContent labelContent = label.getContent();
        DataPartInfo dataPartInfo = new DataPartInfo("testdata");
        labelContent.addDataPartInfo(dataPartInfo);
        out.writeLabel(label);

        // write data part of type testdata
        DataPartEntry dataPartEntry = new DataPartEntry();
        out.putNextDataPartEntry(dataPartEntry);
        out.write(testData);

        // finish sending
        out.close();

        // read local ID from response
        String localId = (String)out.getShsHeaders().get("localid");
        System.out.println("Message sent. txId=" + txId + " localId=" + localId);
    }

. . .
```

6.1.2 List SHS messages

```
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.ProtocolException;
import java.util.Iterator;
import java.util.List;import se.statskontoret.shsapi.*;

. . .

    public void listShsMessages()
        throws MalformedURLException, ShsConnectionException,
               ProtocolException, SSLException, IOException
    {
         // receiver actor ID
        String to = "5544331234";

        // product type ID
        String productType = "9ddfd1cd-11ee-faa6-3224-f192e50614b0";

        // SHS distribution service URL
        String fetchServiceUrl = "https://shs.node.se:11292/ds";

        // create service object for communicating with SHS
        ShsFetchService fetchService =
            ShsFactory.createFetchService(fetchServiceUrl);

        // create filter for message search
        MessageInfoFilter messageInfoFilter = new MessageInfoFilter();
        messageInfoFilter.addProductType(productType);

        // retrieve a list of fetchable messages with the given product type
        List list = fetchService.getMessageInfoList(to, messageInfoFilter);

        // print some info about the messages
        for (Iterator i = list.iterator() ; i.hasNext() ; )
        {
            MessageInfo info = (MessageInfo)i.next();

            System.out.println(info.getTxId() + " " + info.getFrom().getId() + " " +
                               info.getTo().getId());
        }
    }

. . .
```

6.1.3 Fetch an SHS message

```
import java.io.IOException;
import java.net.MalformedURLException;
import se.statskontoret.shsapi.*;

. . .

    public void fetchShsMessage()
        throws MalformedURLException, ShsConnectionException, IOException
    {
         // receiver actor ID
        String to = "5544331234";

        // txId of message to fetch
        String txId = "001700dc-4f94-22c9-0000-00fce034012d";

        // SHS distribution service URL
        String fetchServiceUrl = "https://shs.node.se:11292/ds";

        // create service object for communicating with SHS
        ShsFetchService fetchService =
            ShsFactory.createFetchService(fetchServiceUrl);

        // start fetching message
        MessageInputStream in =
            fetchService.getMessageInputStream(to, txId);

        // read label and print some info from it
        Label label = in.getLabel();
        System.out.println("From: " + label.getFrom().getId());
        System.out.println("Subject: " + label.getSubject());

        // read all data parts and print them
        DataPartEntry dataPartEntry;
        while ((dataPartEntry = in.getNextDataPartEntry()) != null)
        {
            System.out.println("Data:");

            byte[] buf = new byte[1000];
            int n;
            while ((n = in.read(buf)) > 0)
            {
                System.out.write(buf, 0, n);
            }
        }
        in.close();

        // acknowledge the fetching
        fetchService.acknowledgeMessage(to, txId);
    }

. . .
```

6.1.4 Send a synchronous SHS message

```
import java.io.IOException;
import java.net.MalformedURLException;
import se.statskontoret.shsapi.*;

. . .

    public void sendSyncShsMessage()
        throws MalformedURLException, ShsConnectionException, IOException
    {
        // sender actor ID
        String from = "5544331234";

         // receiver actor ID
        String to = "5566771234";

        // product type ID
        String productType = "3aba08f5-769a-fec7-46a4-c516bb300061";

        // SHS receiving URL
        String sendServiceUrl = "https://shs.node.se:11292/rs";

        // data part content
        byte[] testData = "<test>This is test data</test>\r\n".getBytes();

        // create service object for communicating with SHS
        ShsSendService sendService =
            ShsFactory.createSendService(sendServiceUrl);

        // create stream for sending an SHS message
        SyncMessageOutputStream out =
            sendService.createSyncMessageOutputStream();

        // create and write label
        Label label = new Label(from, to, productType);
        LabelContent labelContent = label.getContent();
        DataPartInfo dataPartInfo = new DataPartInfo("testdata");
        labelContent.addDataPartInfo(dataPartInfo);
        out.writeLabel(label);

        // write data part of type testdata
        DataPartEntry dataPartEntry = new DataPartEntry();
        out.putNextDataPartEntry(dataPartEntry);
        out.write(testData);

        // finish sending
        out.close();

        // read response
        MessageInputStream in = out.getResponse();

        // read first data part and print it
        dataPartEntry = in.getNextDataPartEntry();

        System.out.println("Response:");
        byte[] buf = new byte[1000];
        int n;
        while ((n = in.read(buf)) > 0)
        {
            System.out.write(buf, 0, n);
        }
        in.close();
    }
```