Open Mobile API Specification

V3.0

Table of Contents

Table of Figures

simalliance

Table of Tables

1. Terminology

The given terminology is used in this document.

1.1 Abbreviations and Notations

Table 1-1: Abbreviations and Notations

Abbreviation	Description
SE	Secure Element
API	Application Programming Interface
ATR	Answer to Reset (as per ISO/IEC 7816-4)
APDU	Application Protocol Data Unit (as per ISO/IEC 7816-4)
ISO	International Organization for Standardization
ASSD	Advanced Security SD cards (SD memory cards with an embedded security system) as specified by the SD Association
OS	Operating System
RIL	Radio Interface Layer
SFI	Short File ID
FID	File ID
FCP	File Control Parameters
MF	Master File
DF	Dedicated File
EF	Elementary File
OID	Object Identifier
DER	Distinguished Encoding Rules of ASN.1
ASN.1	Abstract Syntax Notation One
OMA	Open Mobile Alliance
DM	Device Management

1.2 Terms

Table 1-2: Terms

Term	Description
Secure Element	A Secure Element (SE) is a tamper-resistant component which is used to provide the security, confidentiality, and multiple application environments required to support various business models. For example UICC/SIM, embedded Secure Element and Secure SD card.
Applet	A general term for a SE application. An application as described in [1] which is installed in the SE and runs within the SE. For example a JavaCard™ application or a native application.
Application	Device/terminal/mobile application. An application which is installed in the mobile device and runs within the mobile device.
Session	An open connection between an application on the device (e.g. mobile phone) and a SE.
Channel	An open connection between an application on the device (e.g. mobile phone) and an applet on the SE.

sim alliance

2. Informative References

Table 2-1: Informative References

Specification	Description
[1] GP v2.2	GlobalPlatform Card Specification v2.2
[2] ISO/IEC 7816-4:2005	Identification cards - Integrated circuit cards - Part 4: Organisation, security and commands for interchange
[3] ISO/IEC 7816-5:2004	Identification cards - Integrated circuit cards - Part 5: Registration of application providers
[4] ISO/IEC 7816-15:2004	Identification cards - Integrated circuit cards with contacts - Part 15: Cryptographic information application
[5] PKCS #11 v2.20	Cryptographic Token Interface Standard Go to following website for PKCS#15 documentation: http://www.rsa.com/rsalabs/node.asp?id=2133
[6] PKCS #15 v1.1	Cryptographic Token Information Syntax Standard
[7] Java™ Cryptography Architecture API Specification & Reference	Go to the following website for JCA documentation: http://download.oracle.com/javase/1.4.2/docs/guide/security/CryptoSpec.html
[8] ISO/IEC 8825-1:2002 | ITU-T Recommendation X.690 (2002)	Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)
[9] GlobalPlatform Secure Element Access Control, v1.0	Specification for controlling access to SEs based on access policies that are stored in the SEs

simalliance

3.   Overview

The API specified in this document enables mobile applications to access different SEs in mobile devices, such as SIMs or embedded SEs.

This specification provides interface definitions and UML diagrams to allow the implementation on various mobile platforms and in different programming languages.

If namespace is supported by the programming language, it shall be org.simalliance.openmobileapi, For the procedural interface the prefix "`OMAPI_`" is used instead.

4. Architecture

The following picture provides an overview of the Open Mobile API architecture.
The architecture is divided into three functional layers:

- The Transport Layer is the foundation for the Service Layer APIs. It provides general access to SEs when an application is accessing it via the generic Transport API. The Transport Layer uses APDUs to access a SE (see chapter 6 for details).

- The Service Layer provides a more abstract interface to various functions on the SE. They will be much easier to use by application developers than the generic transport API. One example could be an email application that uses a sign() function of the Crypto API and which lets the Crypto API do all the APDU exchange with the SE (rather than handle all the required APDUs directly in the email application).

- The Application Layer represents the various applications that make use of the Open Mobile API.
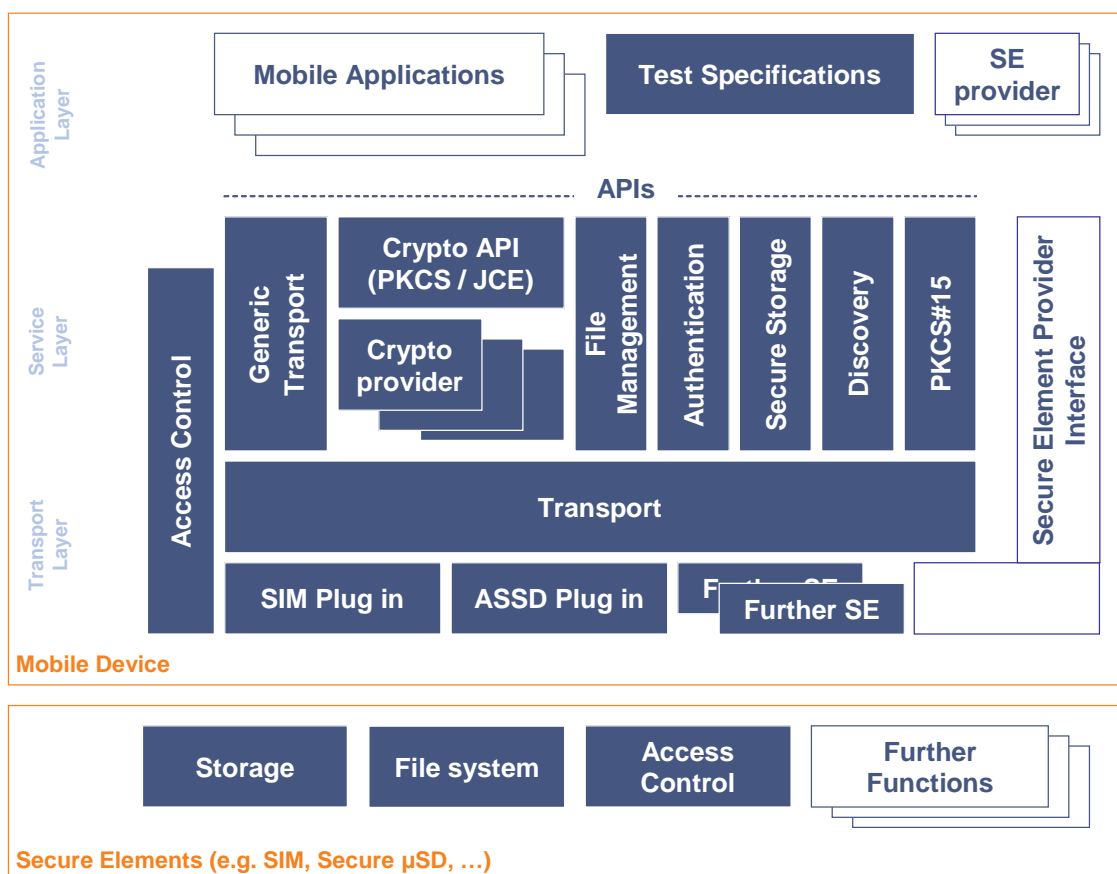
Figure 4-1: Architecture overview

The architecture can be mapped to different OS and might look different depending on the OS.
The description of the APIs uses an abstraction level that allows an easy implementation in different programming languages (e.g. Java or C++).

5. API Description

In general the SIMalliance Open Mobile API defines an interface which enables a software platform that supports object oriented concepts, such as Exceptions or objects and instances, to access a SE. For the Transport API, chapter 6.2 defines an additional interface that is intended for software platforms where such an interface is not available, e.g. an ANSI-C platform.

The interface and data types are not bound to a specific software platform or programming language but defined through a logical type that can be mapped accordingly to the corresponding platform representation.

The following types are used to describe return values, parameters and errors. If supported by the platform, errors may be mapped to exceptions.

Value types

Boolean:	A primitive type, can be true or false.
Int:	A primitive type, mapped to the integer of the platform.
Byte[]:	An array of single byte (8 bits) values.
String:	A string of characters.
Context:	An object representing the execution context of an application (only for object oriented languages).
Void:	Not a type, indicates that the method has no return value.
RESULT:	Return value of a function call. Typically mapped into a primitive 'int' data value (only for procedural interface).
Handle:	A handle represents the connection towards a specific reader or session or channel. Typically mapped into a primitive 'int' or 'long' data value. For security reasons, the handles should be random enough in order to avoid brute-force guessing by other applications (only for procedural interface).

Error/return types:

Success:	No error was encountered (only for procedural interface).
NullPointerError:	Raised when NULL is given where data is required.
IllegalParameterError:	Raised when a method is given an incorrect parameter (e.g. bad format for an APDU).
IllegalStateError:	Raised when used in the wrong context (e.g. being closed).
SecurityError:	An error related to security conditions not being satisfied.
ChannelNotAvailableError:	An error occurs if the basic channel is blocked/busy or if there is no more logical channel available.
NoSuchElementError:	Raised when an AID is not available.
IllegalReferenceError:	An error occurs if the reference cannot be found.
OperationNotSupportedError:	An error occurs if the operation is not supported.
IOError:	An error related to communication (I/O).
GeneralError:	A general error occurred - no further diagnosis (only for procedural interface).

The methods are described as followed:
<return value type> <method name> (<parameter1 type> <parameter1 name> …)

6. Transport API

The Transport API, as part of the Open Mobile API, provides a communication framework to SEs available in the Open Mobile device.

6.1 Overview

The role of the Transport API is to provide the means for applications to access the SE(s) available on the device. The access provided is based on the concepts defined by ISO/IEC 7816-4:

- APDUs: the format of the messages which are exchanged with the SE, basically a byte array, is sent to the SE (or more precisely to an applet in the SE) and the SE responds with another byte array. For details of the exact formatting of such byte arrays, refer to ISO/IEC 7816-4.
- Basic and Logical Channels: the communication abstraction to the SE: Channels are the way to transmit APDUs, and can be opened simultaneously (although APDUs can be sent one at a time, by waiting for the response before sending the next APDU).

This API relies on a 'connection' pattern. The client application (running on the device connected to the SE, e.g. the phone) opens a connection to the SE (a 'session') and then opens a logical or basic channel to an applet running in the SE.

On top of this pattern, there are a number of constraints that are enforced by the system.
An application cannot send 'channel management' APDUs on its own, as this would break the isolation feature given by the logical channel. Once a channel is opened, it is allocated to communicate with one and only one applet in the SE. In the same manner, the SELECT by DF name APDU cannot be sent by the terminal application.
The restrictions for the system should be implemented in the modules that are directly handling the communication with the SE and not in the API itself to ensure that attackers cannot overcome the APDU filters. Thus if possible, the baseband should take care of the filtering or at least the RIL that communicates with the baseband.
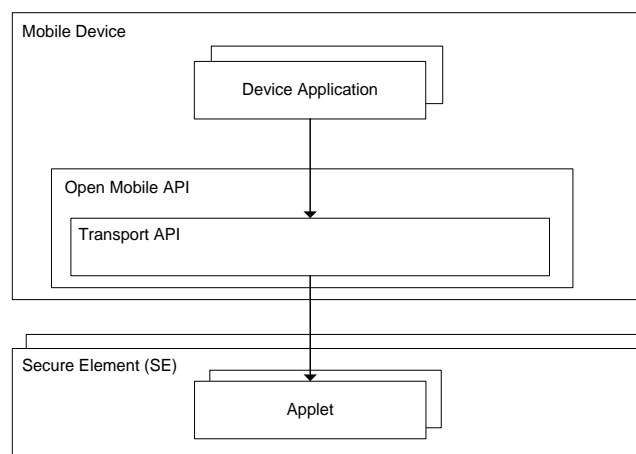
Figure 6-1: Transport API overview

6.2 Object interface

This class diagram contains all classes of the Transport API. The SEService class realizes a connector to the SE framework system and can be used to retrieve all SE readers available in the

system. The Reader class can be used to access the SE connected with the selected reader. The Session class represents a session to an SE established by the reader and allows different communication channels to be opened represented by the Channel class.

org.simalliance.openmobileapi
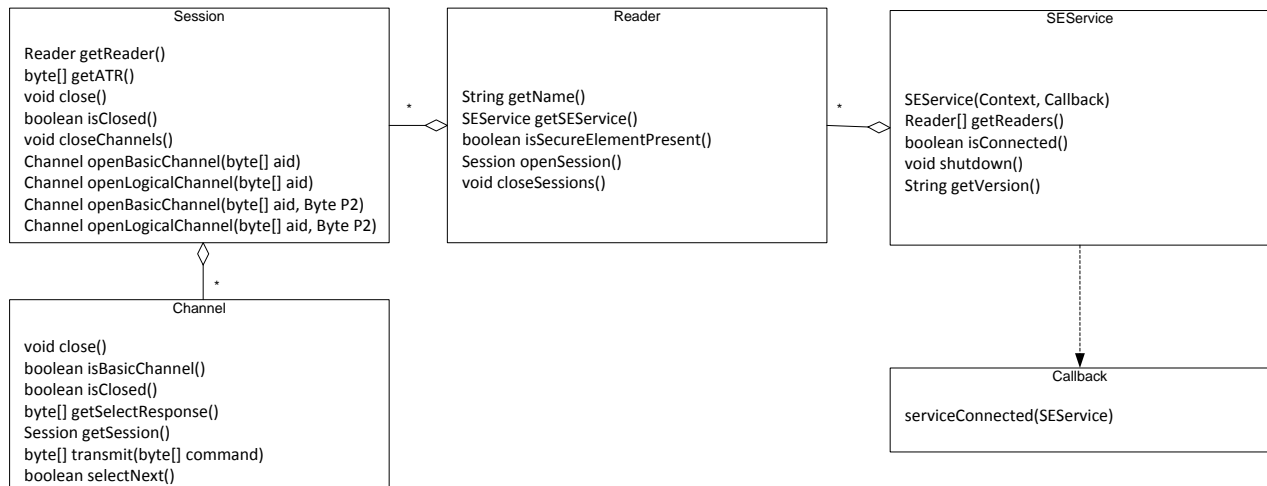
Figure 6-2: Transport API class diagram

6.2.1   Usage pattern

The usage pattern of the Transport API is as follows:

1. The application gets access to the SE service(s):
    It creates an SEService class, passing an object implementing the SEService.Callback interface, whose serviceConnected method is called asynchronously when the connection is established.
    This does not represent a connection with the SE itself, but with the subsystem implementing the SE access functionality.

2. The application enumerates the available readers.
    Readers are the slots where SEs are connected (in a removable or non-removable manner). Once the user or an application-specific algorithm has selected a Reader, then the application opens a session on this reader.

3. With this session, the application can retrieve the ATR of the SE, and if it matches with one of the known ATRs, it can start opening channels with applets in the SE.

4. To open a channel, the application will use the AID of the applet or use the default applet on the newly opened channel.

5. Then the terminal application can start transmitting APDUs to the applet.

6. Once done, the application can close any existing channels or even sessions, and its connection to the SEService.

6.2.2   Class: SEService

The SEService realizes the communication to available SEs on the device.

This is the entry point of this API. It is used to connect to the infrastructure and get access to a list of SE readers.

(a) Constructor: SEService(Context context, SEService.CallBack listener)

Establishes a new connection that can be used to connect to all the SEs available in the system. The connection process can be quite long, so it happens in an asynchronous way. It is usable only if the specified listener is called or if isConnected() returns true.

The call-back object passed as a parameter will have its serviceConnected() method called when the connection actually happen.

Parameters:

context - the context of the calling application. Cannot be null.

listener - a SEService.CallBack object. Can be null.

(b) Method: Reader[] getReaders()

Returns the list of available SE readers. There must be no duplicated objects in the returned list. All available readers shall be listed even if no card is inserted.

Return value:

The readers list, as an array of readers. If there are no readers the returned array is of length 0.

Errors:

NullPointerError – if context is NULL.

IllegalStateError - if the SEService object is not connected.

(c) Method: boolean isConnected()

Tells whether or not the service is connected.

Return value:

True if the service is connected.

(d) Method: void shutdown()

Releases all SE resources allocated by this SEService (including any binding to an underlying service). As a result isConnected() will return false after shutdown() was called. After this method call, the SEService object is not connected.

It is recommended to call this method in the termination method of the calling application (or part of this application) which is bound to this SEService.

(e) Method: String getVersion()

Returns the version of the Open Mobile API Specification this implementation is based on.

Return value:

String containing the Open Mobile API version (e.g. "3.0").

6.2.3   Class (or interface): SEService:CallBack

Interface to receive call-backs when the service is connected.

If the target language and environment allows it, then this shall be an inner interface of the SEService class.

(a) Method: void serviceConnected(SEService service)

Called by the framework when the service is connected.

Parameters:

service - the connected service.

6.2.4 Class: Reader

Instances of this class represent SE readers supported by this device. These readers can be physical devices or virtual devices. They can be removable or not. They can contain one SE that can or cannot be removed.

(a) Method: String getName()

Return the name of this reader.

- If this reader is a SIM reader, then its name must be "SIM[Slot]"
- If the reader is a SD or micro SD reader, then its name must be "SD[slot]"
- If the reader is an embedded SE reader, then its name must be "eSE[slot]"

Slot is a decimal number without leading zeros. The numbering must start with 1 (e.g. SIM1, SIM2, … or SD1, SD2, … or eSE1, eSE2, …). The slot number "1" for a reader is optional (SIM and SIM1 are both valid for the first SIM-reader, but if there are two readers then the second reader must be named SIM2). This applies also for other SD or SE readers.

Return value:

The reader name, as a String.

(b) Method: SEService getSEService()

Return the SE service this reader is bound to.

Return value:

The SEService object.

(c) Method: boolean isSecureElementPresent()

Check if a SE is present in this reader.

Return value:

True if the SE is present, false otherwise.

(d) Method: Session openSession()

Connects to a SE in this reader.

This method prepares (initialises) the SE for communication before the session object is returned (i.e. powers the SE by ICC ON if it is not already on).

There might be multiple sessions opened at the same time on the same reader. The system ensures the interleaving of APDUs between the respective sessions.

Return value:

A Session object to be used to create channels.

Errors:

IOError - if something went wrong when communicating with the SE or the reader.

(e) Method: void closeSessions()

Close all the sessions opened on this reader. All the channels opened by all these sessions will be closed.

6.2.5 Class: Session

Instances of this class represent a connection session to one of the SEs available on the device. These objects can be used to get a communication channel with an applet in the SE. This channel can be the basic channel or a logical channel.

(a) Method: Reader getReader()

Get the reader that provides this session.

Return value:

The reader object.

(b) Method: byte[] getATR()

Get the ATR of this SE.

The returned byte array can be null if the ATR for this SE is not available.

Return value:

The ATR as a byte array or null.

(c) Method: void close()

Close the connection with the SE. This will close any channels opened by this application with this SE.

(d) Method: boolean isClosed()

Tells if this session is closed.

Return value:

True if the session is closed, false otherwise.

(e) Method: void closeChannels()

Close any channel opened on this session.

(f) Method: Channel openBasicChannel(byte[] aid, Byte P2)

Get access to the basic channel, as defined in the ISO/IEC 7816-4 specification (the one that has number 0). The obtained object is an instance of the channel class.

If the AID is null, it means no applet is to be selected on this channel and the default applet is used. If the AID is defined then the corresponding applet is selected.

Once this channel has been opened by a device application, it is considered as 'locked' by this device application, and other calls to this method will return null, until the channel is closed. Some SEs (like the UICC) might always keep the basic channel locked (i.e. return null to applications), to prevent access to the basic channel, while some others might return a channel object implementing some kind of filtering on the commands, restricting the set of accepted command to a smaller set.

It is recommended for the UICC to reject the opening of the basic channel to a specific applet, by always answering null to such a request.

For other SEs, the recommendation is to accept opening the basic channel on the default applet until another applet is selected on the basic channel. As there is no other way than a reset to select again the default applet, the implementation of the transport API should guarantee that the openBasicChannel(null) command will return null until a reset occurs. With the previous release (V2.05), it was not possible to set P2 value, this value was always set to '00'. Except for specific needs it is recommended to keep P2 set to '00'. It is recommended that the device allows all values for P2, however only the following values are mandatory: '00', '04', '08', '0C' (as defined in [2]).

The implementation of the underlying SELECT command within this method shall be based on ISO 7816-4 with following options:

CLA = '00'

INS = 'A4'

P1='04' (Select by DF name/application identifier)

The select response data can be retrieved with byte[] getSelectResponse().

The API shall handle received status word as follows. If the status word indicates that the SE was able to open a channel (e.g. status word '90 00' or status words referencing a warning in ISO-7816-4: '62 XX" or "63 XX') the API shall keep the channel opened and the next getSelectResponse() shall return the received status word.

Other received status codes indicating that the SE was not able to open a channel shall be considered as an error and the corresponding channel shall not be opened.
The function without P2 as parameter is provided for backwards compatibility and will fall back to a select command with P2='00'.

Parameters:
aid - the AID of the applet to be selected on this channel, as a byte array, or null if no applet is to be selected.
P2 - the P2 parameter of the SELECT APDU executed on this channel.

Return value:
An instance of channel if available or null.

Errors:
IllegalParameterError - if the aid's length is not within 5 to 16 (inclusive).
IllegalStateError - if the SE session is used after being closed.
SecurityError - if the calling application cannot be granted access to this AID or the default applet on this session.
NoSuchElementError – If the AID on the SE is not available or cannot be selected.
OperationNotSupportedError – if the given P2 parameter is not supported by the device.
IOError - if there is a communication problem to the reader or the SE.

(g) Method: Channel openBasicChannel(byte[] aid)
This method is provided to ease the development of mobile application and for compliancy with existing applications. This method is equivalent to openBasicChannel(aid, P2=0x00).

(h) Method: Channel openLogicalChannel(byte[] aid, Byte P2)
Open a logical channel with the SE, selecting the applet represented by the given AID. If the AID is null, which means no applet is to be selected on this channel, the default applet is used. It's up to the SE to choose which logical channel will be used.
With the previous release (V2.05) it was not possible to set P2 value, this value was always set to '00'. Except for specific needs it is recommended to keep P2 set to '00'. It is recommended that the device allows all values for P2, however only the following values are mandatory: '00', '04', '08', '0C' (as defined in [2]).

The implementation of the underlying SELECT command within this method shall be based on ISO 7816-4 with the following options:
CLA = '01' to '03', '40 to 4F'
INS = 'A4'
P1='04' (Select by DF name/application identifier)

The select response data can be retrieved with byte[] getSelectResponse().

The API shall handle received status word as follows. If the status word indicates that the SE was able to open a channel (e.g. status word '90 00' or status words referencing a warning in ISO-7816-4: '62 XX" or "63 XX') the API shall keep the channel opened and the next getSelectResponse() shall return the received status word.
Other received status codes indicating that the SE was not able to open a channel shall be considered as an error and the corresponding channel shall not be opened.
In the case of a UICC it is recommended that the API rejects the opening of the logical channel without a specific AID, by always answering null to such a request.

The function without P2 as parameter is provided for backwards compatibility and will fall back to a select command with P2=00.

Parameters:
aid - the AID of the applet to be selected on this channel, as a byte array.
P2 - the P2 parameter of the SELECT APDU executed on this channel.

Return value:
An instance of channel. Null if the SE is unable to provide a new logical channel even if this channel would first be used by the implementation to retrieve the Access Control rules according to the GlobalPlatform Secure Element Access Control Specification.

Errors:
IllegalParameterError - if the aid's length is not within 5 to 16 (inclusive).
IllegalStateError - if the SE is used after being closed.
SecurityError - if the calling application cannot be granted access to this AID or the default applet on this session.
Note, failure in retrieving rules due to the lack of a new logical channel (and only this failure) should result in a Null return value and not a security exception. This is in line with the GlobalPlatform Secure Element Access Control Specification, as the access to the SE applet will be anyway denied.
NoSuchElementError – If the AID on the SE is not available or cannot be selected or a logical channel is already open to a non-multiselectable applet.
OperationNotSupportedError – if the given P2 parameter is not supported by the device.
IOError - if there is a communication problem to the reader or the SE.

(i) Method: Channel openLogicalChannel(byte[] aid)
This method is provided to ease the development of mobile applications and for compliancy with existing applications. This method is equivalent to openLogicalChannel(aid, P2=0x00).

6.2.6   Class: Channel
Instances of this class represent an ISO/IEC 7816-4 channel opened to a SE. It can be either a logical channel or the basic channel.
They can be used to send APDUs to the SE. Channels are opened by calling the Session.openBasicChannel(byte[]) or Session.openLogicalChannel(byte[]) methods.

(a) Method: void close()
Closes this channel to the SE. If the method is called when the channel is already closed, this method will be ignored.
The close() method shall wait for completion of any pending transmit(byte[] command) before closing the channel.

(b) Method: boolean isBasicChannel()
Returns a boolean telling if this channel is the basic channel.

Return value:
True if this channel is a basic channel.
False if this channel is a logical channel.

(c) Method: boolean isClosed()
Tells if this channel is closed.

Return value:
True if the channel is closed, false otherwise.

(d) Method: byte[] getSelectResponse()

Returns the data as received from the application select command inclusively the status word received at applet selection.
The returned byte array contains the data bytes in the following order:
[<first data byte>, …, <last data byte>, <sw1>, <sw2>]

Return value:
- The data as returned by the application select command inclusive of the status word.
- Only the status word if the application select command has no returned data.
- Null if an application select command has not been performed or the selection response cannot be retrieved by the reader implementation.

(e) Method: Session getSession()
Get the session that has opened this channel.

Return value:
The session object this channel is bound to.

(f) Method: byte[] transmit(byte[] command)
Transmit an APDU command (as per ISO/IEC 7816-4) to the SE. The underlying layers generate as many TPDUs as necessary to transport this APDU. The API shall ensure that all available data returned from the SE, including concatenated responses, are retrieved and made available to the calling application. If a warning status code is received the API won't check for further response data but will return all data received so far and the warning status code.
The transport part is invisible from the application. The generated response is the response of the APDU which means that all protocol related responses are handled inside the API or the underlying implementation.

The transmit method shall support extended length APDU commands independently of the coding within the ATR.

For status word '61 XX' the API or underlying implementation shall issue a GET RESPONSE command as specified by ISO 7816-4 standard with LE=XX; this includes the scenario where the SE returns overall response data of more than 256 bytes to the APDU. For the status word '6C XX', the API or underlying implementation shall reissue the input command with LE=XX. For other status words (including warning status words), the API (or underlying implementation) shall return the complete response including all response data and final status word to the device application. Note that the handling of GET RESPONSE specified above implies that the API or underlying implementation shall handle response data of more than 256 bytes which is returned by the SE.

The API (or underlying implementation) shall not handle the received status words internally. The channel shall not be closed even if the SE answered with an error code.
The system ensures the synchronization between all the concurrent calls to this method, and that only one APDU will be sent at a time, irrespective of the number of TPDUs that might be required to transport it to the SE. The entire APDU communication to this SE is locked to the APDU.
The channel information in the class byte in the APDU will be ignored. The system will add any required information to ensure the APDU is transported on this channel.
The only restrictions on the set of commands that can be sent is defined below, the API implementation shall be able to send all other commands:
- MANAGE_CHANNEL commands are not allowed.
- SELECT by DF Name (P1=04) are not allowed.
- CLA bytes with channel numbers are de-masked.

Parameters:
command - the APDU command to be transmitted, as a byte array.

Return value:
The response received, as a byte array. The returned byte array contains the data bytes in the following order: [<first data byte>, …, <last data byte>, <sw1>, <sw2>]

Errors:
NullPointerError – if command is NULL.
IllegalParameterError – if:
- the command byte array is less than 4 bytes long, or
- Lc byte is inconsistent with the length of the byte array, or
- CLA byte is invalid according to [2] (0xff), or
- INS byte is invalid according to [2] (0x6x or 0x9x).

IllegalStateError - if the channel is used after being closed.
SecurityError - if the command is filtered by the security policy.
IOError - if there is a communication problem to the reader or the SE.

(g) Method: boolean selectNext()

Performs a selection of the next Applet on this channel that matches to the partial AID specified in the openBasicChannel(byte[] aid) or openLogicalChannel(byte[] aid) method. This mechanism can be used by a device application to iterate through all applets matching to the same partial AID.

If selectNext() returns true, a new applet was successfully selected on this channel. If no further applet exists with matches to the partial AID, this method returns false and the already selected applet stays selected.

Since the API cannot distinguish between a partial and full AID, the API shall rely on the response of the SE for the return value of this method.

The implementation of the underlying SELECT command within this method shall use the same values as the corresponding openBasicChannel(byte[] aid) or openLogicalChannel(byte[] aid) command with the option:
P2='02' (Next occurrence)

The select response stored in the channel object, shall be updated with the APDU response of the SELECT command.

Return value:
True if a new applet was selected on this channel.
False if the already selected applet stays selected on this channel.

Errors:
IllegalStateError - if the SE is used after being closed.
OperationNotSupportedError - if this operation is not supported by the card.
IOError - if there is a communication problem to the reader or the SE.

6.3 Procedural interface

This interface is based on the object interface but adapted for platforms where concepts like objects or exceptions are not available.

The overall design of the Object interface is kept with the separation of readers, sessions and channels. However, for the ease of the interface, the concept of an underlying service (represented by SEService) is removed. All exceptions and error conditions are mapped into result codes.

The behaviour of the functions defined in the procedural interface is the same as their counterparts in the object interface.

OMAPI_

SEService

OMAPI RESULT omapi_get_version(String pVersion, Int *pLength)
OMAPI RESULT omapi_get_readers(Handle *phReaders, Int *pLength)

Reader

OMAPI RESULT omapi_reader_get_name(Handle hReader, String pReader, Int *pLength)
OMAPI RESULT omapi_reader_is_secure_element_present(Handle hReader, Boolean *pIsPresent)
OMAPI RESULT omapi_reader_open_session(Handle hReader, Handle *phSession)
OMAPI RESULT omapi_reader_close_sessions(Handle hReader)

Session

OMAPI RESULT omapi_session_get_reader(Handle hSession, Handle *phReader)
OMAPI RESULT omapi_session_get_atr(Handle hSession, Byte *pAtr, Int *pLength)
OMAPI RESULT omapi_session_close(Handle hSession)
OMAPI RESULT omapi_session_is_closed(Handle hSession, Boolean *pIsClosed)
OMAPI RESULT omapi_session_close_channels(Handle hSession)
OMAPI RESULT omapi_session_open_basic_channel(Handle hSession, Byte *AID, Int length, Byte P2, Handle *phChannel)
OMAPI RESULT omapi_session_open_logical_channel(Handle hSession, Byte *AID, Int length, Byte P2, Handle *phChannel)

Channel

OMAPI RESULT omapi_channel_close(Handle hChannel)
OMAPI RESULT omapi_channel_is_basic_channel(Handle hChannel, Boolean *pIsBasicChannel)
OMAPI RESULT omapi_channel_is_closed(Handle hChannel, Boolean *pIsClosed)
OMAPI RESULT omapi_channel_get_select_response(Handle hChannel, Byte *pSelectResponse, Int *pLength)
OMAPI RESULT omapi_channel_get_session(Handle hChannel, Handle *phSession)
OMAPI RESULT omapi_channel_transmit(Handle hChannel, Byte *pCommand, Int cmdLength, Byte *pResponse, Int *pRspLength)
OMAPI RESULT omapi_channel_transmit_get_response(Handle hChannel, Byte *pResponse, Int *pRspLength)
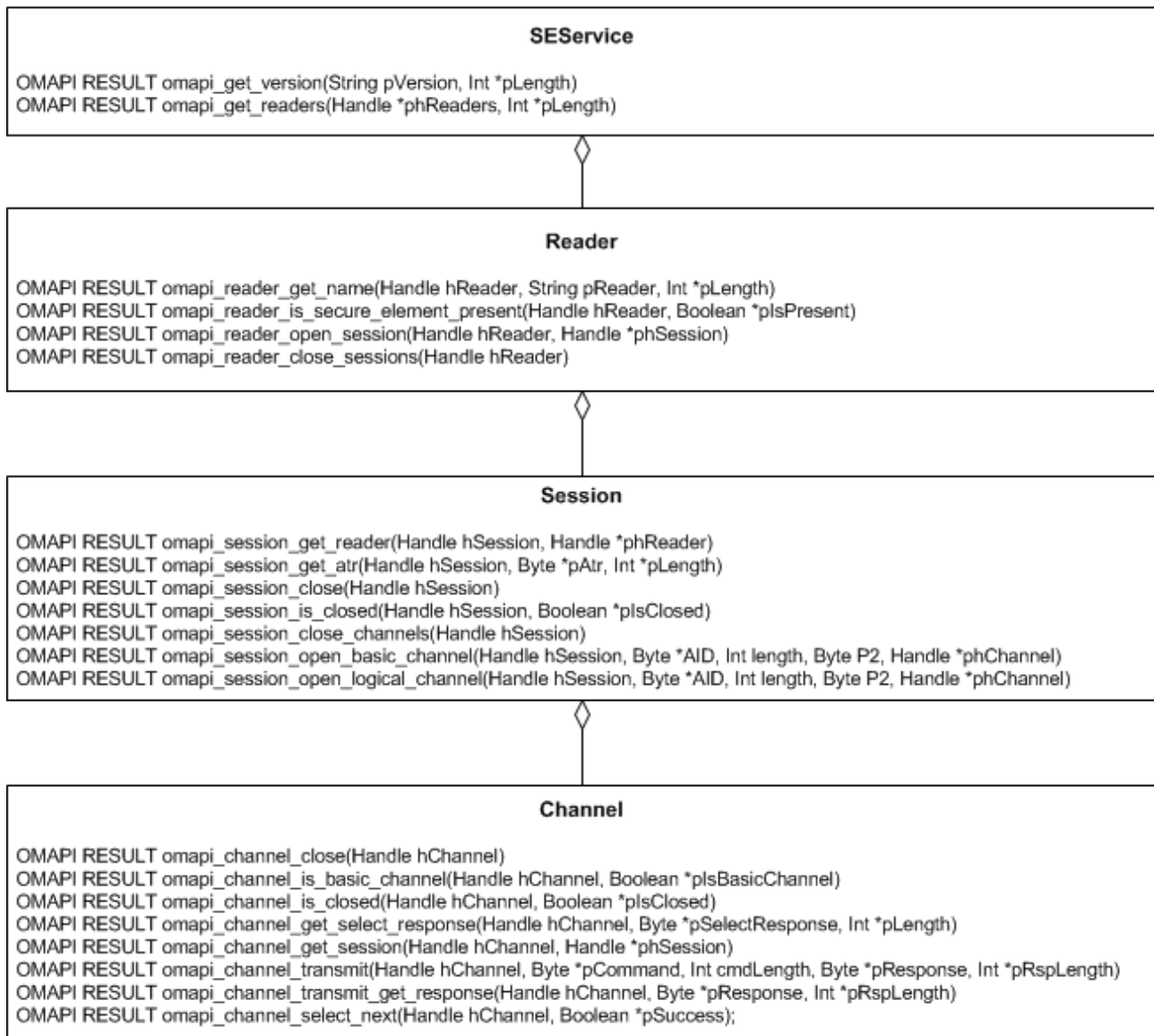OMAPI RESULT omapi_channel_select_next(Handle hChannel, Boolean *pSuccess);

Figure 6-3: Transport API procedure diagram

6.3.1　　　　Usage pattern

When data is to be returned from the procedural interface through a parameter, e.g. the ATR of a session or the select response, the client has to allocate the proper amount of memory.
For such, the client can either

- pre-allocate enough memory and specify the amount of allocated memory in the function call or,
- call the function twice. First call would be with null as parameter to determine the proper amount of memory required and allocated and referenced in the second call.

The omapi_channel_transmit() function works differently as the memory has either to be pre-allocated before the function call or omapi_channel_transmit_receive_response() has to be called to retrieve the actual response APDU. Every call to omapi_channel_transmit() will transmit the supplied APDU.

The client always has to allocate all memory required for the response data of the method call. The methods
- omapi_get_readers() and
- omapi_channel_transmit()
can return part of the response data, e.g. only the first two entries or bytes, and cut off the rest.

The other methods like
- omapi_get_version(),
- omapi_reader_get_name(),
- omapi_session_get_atr(),
- omapi_channel_get_select_response()
can only return all data or nothing.

6.3.2   SEService mapping

(a) OMAPI RESULT omapi_get_version(String pVersion, Int *pLength)

pVersion has to be a properly allocated string or null if the function should return the proper string length in pLength.

Parameters
[out] String *pVersion - allocated string buffer that contains the version string or null to determine the proper length in pLength.
[in|out] Int *pLength - size of version string length.

Return
Success - all went ok.
IllegalParameterError - pLength is too short for the version string to be returned.
GeneralError - general error not further specified.

(b) OMAPI RESULT omapi_get_readers(Handle *phReaders, Int *pLength)

phReaders must be properly allocated or null where the function returns the proper amount of handles in pLength.
When pLength is smaller than the available reader list, only pLength handles are returned in phReaders. If pLength is bigger than the available reader list, only the available reader list is returned in phReaders and pLength contains the proper amount of handles in phReaders.

Parameters

[out] Handle *phReaders - allocated list of handles or null to determine the array length.
[in|out] Int *pLength - amount of handles that are allocated/returned in phReaders.

Return
Success - all went ok.
GeneralError - general error not further specified.

6.3.3   Reader mapping

(a) *OMAPI RESULT omapi_reader_get_name(Handle hReader, String pReader, Int *pLength)*

pReader must be properly allocated or null where the function returns the proper amount of memory to be allocated in pLength.
Depending on the software platform, pLength might include a \0 after the actual reader name if the platform representation of strings are zero-terminated.

Parameters
[in] Handle hReader - handle to the reader.
[out] String *pReader - allocated string to retrieve the name of the reader or null to determine the string length.
[in|out] Int *pLength - size of allocated/returned string.

Return
Success - all went ok.

NullPointerError – pLength is null IllegalParameterError - hReader not a valid reader handle or
pLength < actual length of the reader name.
GeneralError - general error not further specified.

(b) *OMAPI RESULT omapi_reader_is_secure_element_present(Handle hReader, Boolean *pIsPresent)*

Parameters
[in] Handle hReader - handle to the reader.
[out] Boolean *pIsPresent - true or false depending on whether a SE is present in the reader - undefined when an error is returned.

Return
Success - all went ok.
IllegalParameterError - hReader not a valid reader handle.
GeneralError - general error not further specified.

(c) *OMAPI RESULT omapi_reader_open_session(Handle hReader, Handle *phSession)*

Parameters

[in] Handle hReader - handle to the reader.

[out] Handle *phSession - handle to the session created for this reader.

Return

Success - all went ok.

NullPointerError – phSession is null IOError - something went wrong in the communication with the SE.

IllegalParameterError - hReader not a valid reader handle.

GeneralError - general error not further specified.

(d) OMAPI RESULT omapi_reader_close_sessions(Handle hReader)

Parameters

[in] Handle hReader - handle to the reader.

Return

Success - all went ok.

IOError - something went wrong in the communication with the SE.

IllegalParameterError - hReader not a valid reader handle.

GeneralError - general error not further specified.

6.3.4 Session mapping

(a) OMAPI RESULT omapi_session_get_reader(Handle hSession, Handle *phReader)

Parameters

[in] Handle hSession - handle to the session.

[out] Handle *phReader - reader handle that provides this session.

Return

Success - all went ok.

NullPointerError – phReader is null.

IllegalParameterError - hSession not a valid session handle.

GeneralError - general error not further specified.

(b) OMAPI RESULT omapi_session_get_atr(Handle hSession, Byte *pAtr, Int *pLength)

pAtr must be properly allocated or null where the function returns the proper amount of memory to be allocated for pAtr in pLength. If the ATR for this SE is not available the returned length is set to zero and return value is "Success".

Parameters

[in] Handle hSession - handle to the session.

[out] Byte *pAtr - allocated byte array to retrieve the ATR or null to determine the array length in pLength.

[in|out] Int *pLength - size of byte array allocated/returned in pAtr.

Return
Success - all went ok.
NullPointerError – pLength is null.
IllegalParameterError - hSession not a valid session handle or pLength < actual length of ATR.
GeneralError - general error not further specified.

(c) *OMAPI RESULT omapi_session_close(Handle hSession)*

Parameters
[in] Handle hSession - handle to the session.

Return
Success - all went ok.
IOError - communication error with the SE while closing a channel.
IllegalParameterError - hSession not a valid session handle.
GeneralError - general error not further specified.

(d) *OMAPI RESULT omapi_session_is_closed(Handle hSession, Boolean *pIsClosed)*

Parameters
[in] Handle hSession - handle to the session.
[out] Boolean *pIsClosed - true or false depending on the session state.

Return
Success - all went ok.
NullPointerError – pIsClosed is null.
IllegalParameterError - hSession not a valid session reference.
GeneralError - general error not further specified.

(e) *OMAPI RESULT omapi_session_close_channels(Handle hSession)*

Parameters
[in] Handle hSession - handle to the session.

Return
Success - all went ok.
IOError - communication error with the SE while closing a channel.
IllegalParameterError - hSession not a valid session handle.
GeneralError - general error not further specified.

(f) *OMAPI RESULT omapi_session_open_basic_channel(Handle hSession, Byte *AID, Int length, Byte P2, Handle *phChannel)*

Parameters

[in] Handle hSession - handle to the session.

[in] Byte *AID - byte array containing the AID to be selected on this channel.

[in] Int length - size of byte array or 0 when no SELECT should be executed.

[in] Byte P2 - P2 byte of the SELECT command if executed. It is recommended that the device allows all values for P2, however only the following values are mandatory: '00', '04', '08', '0C' (definitions of these values are in [2]).

[out] Handle *phChannel - channel handle of the basic channel.

Return

Success - all went ok.

NullPointerError – phChannel is null.

SecurityError - channel not allowed by access control.

IOError - communication error with the SE.

NoSuchElementError - AID cannot be selected.

IllegalParameterError - hSession not a valid session handle.

ChannelNotAvailableError – if basic channel is blocked or busy.

GeneralError - general error not further specified.

(g) OMAPI RESULT omapi_session_open_logical_channel(Handle hSession, Byte *AID, Int length, Byte P2, Handle *phChannel)

In the case of a UICC it is recommended that the API rejects the opening of the logical channel without a specific AID, by always returning ChannelNotAvailableError to such a request.

Parameters

[in] Handle hSession - handle to the session.

[in] Byte *AID - byte array containing the AID to be selected on this channel.

[in] Int length - size of byte array or 0 when no SELECT should be executed.

[in] Byte P2 - P2 byte of the SELECT command if executed.

[out] Handle *phChannel - channel handle of the new logical channel.

Return

Success - all went ok.

NullPointerError – phChannel is null.

SecurityError - channel not allowed by access control.

IOError - communication error with the SE.

NoSuchElementError - AID cannot be selected.

IllegalParameterError - hSession not a valid session handle.

ChannelNotAvailableError – if there are no more logical channels available.

GeneralError - general error not further specified.

6.3.5   Channel mapping

(a) OMAPI RESULT omapi_channel_close(Handle hChannel)

Parameters

[in] Handle hChannel - handle to the channel.

Return

Success - all went ok.

IOError - communication error with the SE while closing the channel.

IllegalParameterError - hChannel not a valid channel handle.

GeneralError - general error not further specified.

(b) OMAPI RESULT omapi_channel_is_basic_channel(Handle hChannel, Boolean *pIsBasicChannel)

Parameters

[in] Handle hChannel - handle to the channel.

[out] Boolean *pIsBasicChannel - true or false depending on the channel type.

Return

Success - all went ok.

NullPointerError – pIsBasicChannel is null.

IllegalParameterError - hChannel not a valid channel handle.

GeneralError - general error not further specified.

(c) OMAPI RESULT omapi_channel_is_closed(Handle hChannel, Boolean *pIsClosed)

Parameters

[in] Handle hChannel - handle to the channel.

[out] Boolean *pIsClosed - false if the channel is open, true in all other cases.

Return

Success - all went ok.

NullPointerError – pIsClosed is null.

GeneralError - general error not further specified.

(d) OMAPI RESULT omapi_channel_get_select_response(Handle hChannel, Byte *pSelectResponse, Int *pLength)

pSelectResponse must be properly allocated or null where the function returns the proper amount of memory to be allocated for pSelectResponse in pLength.

Parameters

[in] Handle hChannel - handle to the channel.

[out] Byte *pSelectResponse - allocated byte array to retrieve the SELECT response or null to determine the array length.

[in|out] Int *pLength - size of byte array allocated/returned in pSelectResponse.

Return

Success - all went ok.

NullPointerError – pLength is null.

IllegalParameterError - hChannel not a valid channel handle or pLength < actual length of the SELECT response.

GeneralError - general error not further specified.

(e) OMAPI RESULT omapi_channel_get_session(Handle hChannel, Handle *phSession)

Parameters
[in] Handle hChannel - handle to the channel.
[out] Handle *phSession - session handle of this channel.

Return
Success - all went ok.
NullPointerError – phSession is null.
IllegalParameterError - hChannel not a valid channel handle.
GeneralError - general error not further specified.

(f) OMAPI RESULT omapi_channel_transmit(Handle hChannel, Byte *pCommand, Int cmdLength, Byte *pResponse, Int *pRspLength)

Transmits an APDU defined in pCommand to the SE. The response APDU is always cached internally and can be retrieved with omapi_channel_transmit_receive_response() until the data is overwritten after the next omapi_channel_transmit() call.

When memory is allocated in pResponse for the response APDU, the response is directly returned and a following call to omapi_channel_transmit_receive_response() is not required. The length of the response APDU (data plus the status word) is returned in pRspLength.

When pRspLength is smaller than the actual response APDU, only first pRspLength bytes are returned in pResponse. If pRspLength is bigger than the actual response APDU, only the actual response data plus its status word is returned in pResponse and pRspLength contains the proper amount of bytes in pResponse.

Parameters
[in] Handle hChannel - handle to the channel.
[in] Byte *pCommand - command APDU to be send to the SE.
[in] Int cmdLength - size of command APDU.
[out] Byte *pResponse - allocated byte array for the response APDU (data plus status word) or null if only the response length should be returned.
[in|out] Int *pRspLength - size of response APDU (data plus status word) to be retrieved with omapi_channel_transmit_receive_response() or contained in pResponse.

Return
Success - all went ok.
NullPointerError – if pCommand is null.
SecurityError - command not allowed by access control.
IOError - communication error with the SE.
IllegalStateError - channel is already closed to this SE.
IllegalParameterError – if:
  * hChannel not a valid channel handle, or
  * cmdLength < 4, or
  * Lc byte is inconsistent with cmdLength, or
  * CLA byte is invalid according to [2] (0xff), or

- INS byte is invalid according to [2] (0x6x or 0x9x).

GeneralError - general error not further specified.

*(g) OMAPI RESULT omapi_channel_transmit_retrieve_response(Handle hChannel, Byte *pResponse, Int *pRspLength)*

Helper function to retrieve the response APDU of the previous transmit() call. Subsequent transmit() calls overwrite the response APDU. pResponse should be allocated with the amount of bytes returned by the transmit() call.

When pRspLength is smaller than the actual response APDU, only pRspLength bytes are returned in pResponse. If pRspLength is bigger than the actual response APDU, only the actual response data plus its status word is returned in pResponse and pRspLength contains the proper amount of bytes in pResponse.

Parameters

[in] Handle hChannel - handle to the channel.

[out] Byte *pResponse - allocated byte array for the response APDU (data plus status word).

[in|out] Int *pRspLength - size of byte array allocated/returned in pResponse.

Return

Success - all went ok.

IllegalStateError - no APDU response available on this channel, e.g. no transmit executed previously.

IllegalParameterError - hChannel not a valid channel handle.

GeneralError - general error not further specified.

*(h) OMAPI RESULT omapi_channel_select_next(Handle hChannel, Boolean *pSuccess)*

Parameters

[in] Handle hChannel - handle to the channel.

[out] Boolean *pSuccess - True if a new applet was selected on this channel.

False if the already selected applet stays selected on this channel.

Return

Success - all went ok.

IOError - communication error with the SE.

IllegalStateError - channel is already closed to this SE.

IllegalParameterError - hChannel not a valid channel handle.

OperationNotSupportedError - SE does not support the select next command.

GeneralError - general error not further specified.

7. Service Layer APIs

The Service APIs as part of the Open Mobile API provides a framework to access SEs available in the mobile device with high level interfaces.

7.1   Overview

The Open Mobile API contains a Service API on top of the Transport API for providing high level API methods for different purposes. The Service API relies on the Transport API to establish communication with the Applets within the SE. The Service API consists of different APIs specialised for different purposes (e.g. File Management API for file operations). Normally each specialised API requires a counterpart on the SE side (e.g. an SE applet providing a defined set of APDUs).
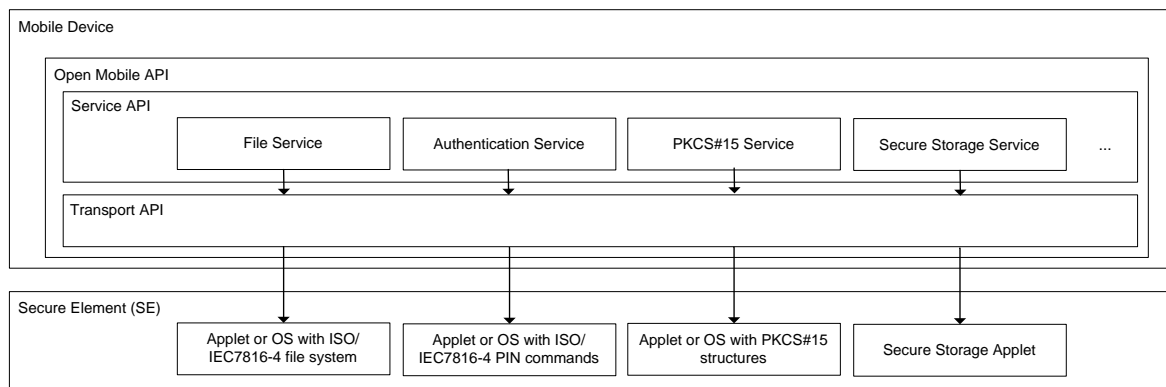
Figure 7-1: Service API overview

Note:

The use of the Service Layer API requires SE access rights (see chapter 10 for more information on SE access rights).

Since all operations within the Service API layer are based on the Transport API, all error conditions of the corresponding transport classes can be thrown in the service layer although they are not explicitly named. E.g. a call to AuthenticationProvider::verifyPin() can cause an IOError because the implementation uses the Channel::transmit() API call internally.

7.2 Class diagram

This class diagram contains the Service API besides the Transport API. The Service API consists of a set of classes derived from the base class Provider. The Crypto API is not part of this diagram, but it is also a component of the Service API. The Crypto API relies on already existing APIs in the OS and realises SE communication via the Transport API.
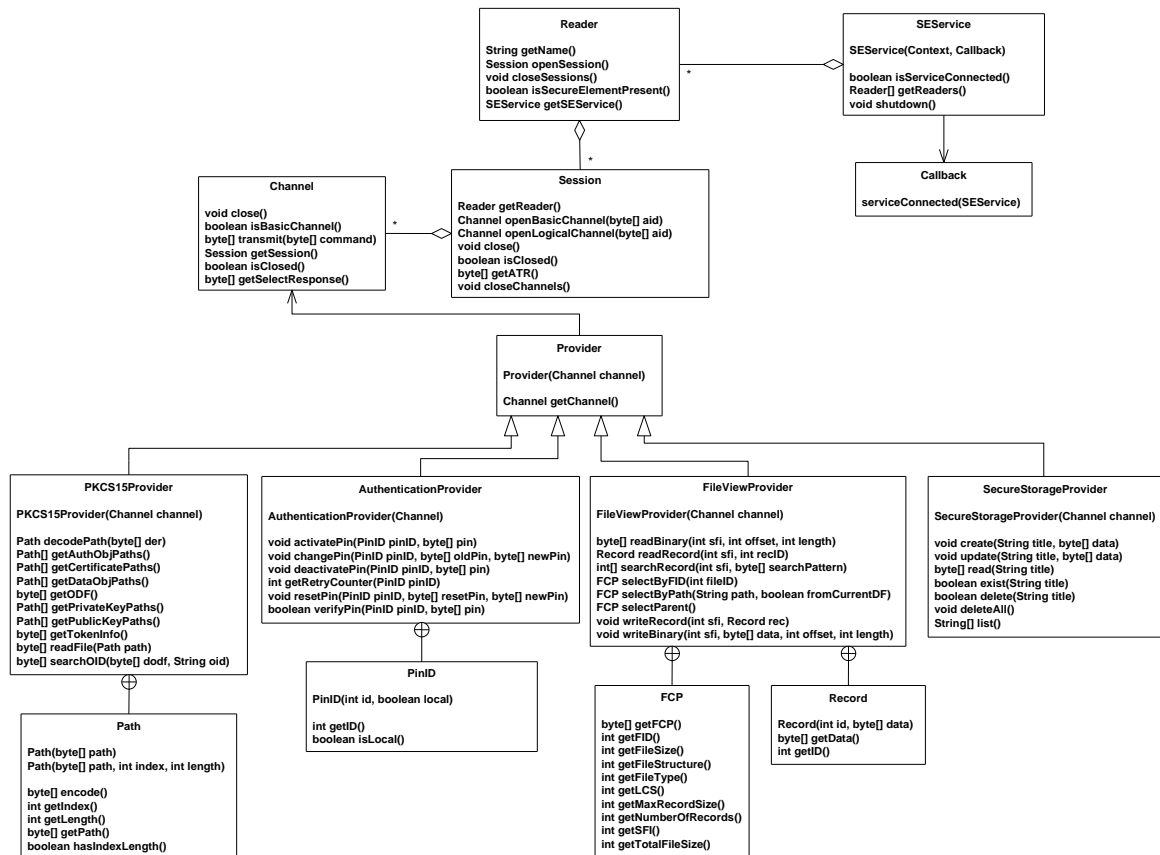
Figure 7-2: Service API class diagram with Provider classes

Besides these Provider classes, the Service API contains a SEDiscovery class which provides a discovery mechanism that can be used for SE selection by defined criteria.

org.simalliance.openmobileapi
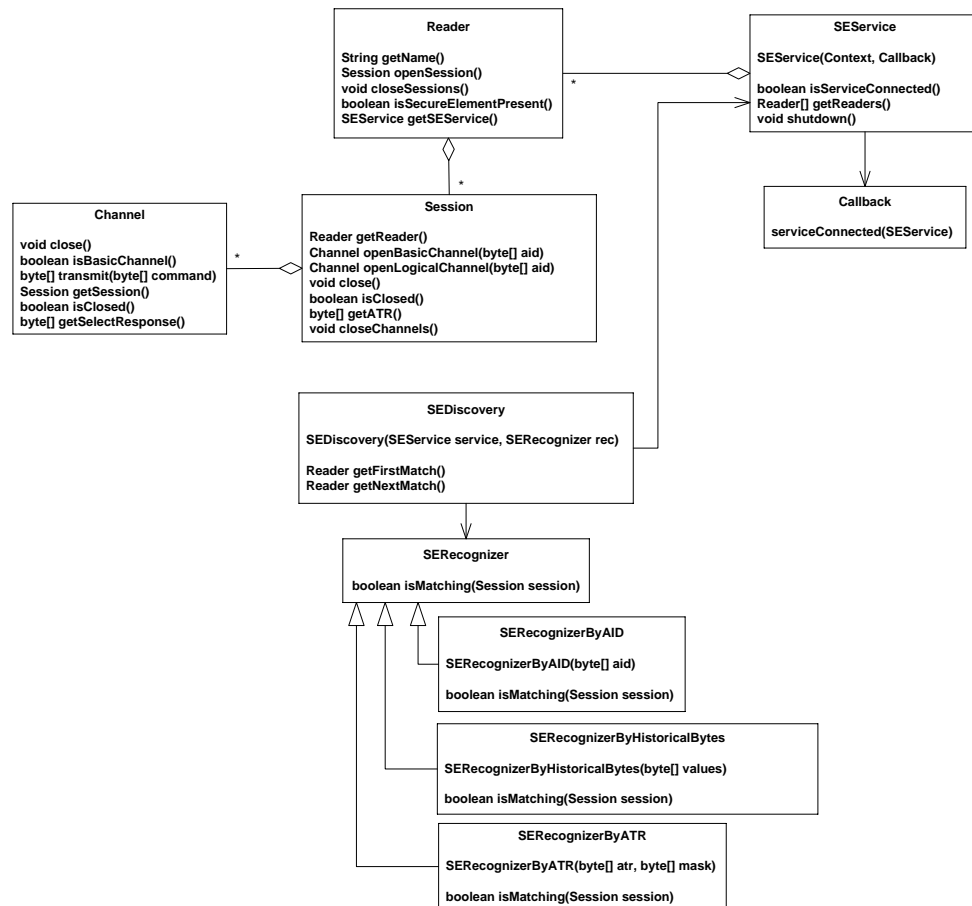
Figure 7-3: Service API class diagram with SEDiscovery classes

7.3   Usage pattern

The usage pattern of the service API is as follows:

1. The application gets access to the SE service(s):
   It creates a SEService class, passing an object implementing the SEService.Callback interface, whose serviceConnected method is called asynchronously when the connection is established. This does not represent a connection with the SE itself, but with the subsystem implementing the SE access functionality.

2. The application enumerates the available readers.
   Readers are the slots where SEs are connected (in a removable or non-removable manner). Once the user or an application-specific algorithm has selected a reader, then the application opens a session on this reader. The right reader can also be chosen by using the Discovery API. The Discovery API allows different criteria to be defined (ATR, AID, …) for finding an appropriate SE and applet in an SE. Finally, the Discovery API allows iterating through the readers containing an SE with the defined criteria.

3. With this session, the application can retrieve the ATR of the SE, and if it matches with one of the known ATRs, it can start opening channels with applets in the SE.

4. To open a channel, the application will use the AID of the applet or use the default applet on the newly opened channel. The application is in charge of selecting an applet which fits to the specific Provider class that will be chosen in the next step for SE operations.

5. The application creates an instance of a certain Provider class depending on the application's intention. If the intention is to read files from the SE's file system the FileViewProvider has to be chosen. The application has to consign the communication channel (established in the step before) to the Provider before the Provider instance can be used for SE operations.

6. Then the terminal application can start performing operations on the selected applet in the SE with the help of the provider's methods. If a FileViewProvider instance was created for reading files from the SE's file system, the application can use the FileViewProvider's methods readBinary() or readRecord().

7. The terminal application can also use several Provider instances alternately on the same channel or it can use the transmit method of the Transport Layer to send any APDUs directly to the SE on that channel. This option is especially useful if the application needs to perform different operations on the same channel. This could happen for example if the application needs to read a file from the SE's file system that requires a successful PIN verification on the same channel as before. In this case, the terminal application can instantiate an Authentication class which provides the verifyPin() method. After the successful PIN verification via the AuthenticationProvider the FileViewProvider can be used to read the file from the SE's file system. Since the SE usually manages the PIN verification individually for each logical channel it is important to apply the AuthenticationProvider and FileViewProvider on the same channel.

8. Once done, the application can close any existing channels or even sessions, and its connection to the SEService.

7.4   Service API Framework

The Open Mobile API provides a set of service layer classes with high level methods for SE operations.

7.4.1   Class: Provider

This Provider class (realised as interface or abstract class) is the base class for all service layer classes. Each service layer class provides a set of methods for a certain aspect (file management, PIN authentication, PKCS#15 structure handling etc) and acts as a provider for service routines. All Provider classes need an open channel for communication with the SE. As such, before a certain Provider class can be used for SE operations, the channel has to be consigned. For performing different operations (PIN authentication, file operation etc) the Provider classes can be easily combined by using the same channel for different Provider classes and alternately calling methods of these different providers. It has to be considered that each Provider class needs a counterpart on the SE side (e.g. an applet with a standardised APDU interface as required by the Provider class). The application using a Provider class for SE interactions is in charge of assigning a channel to the Provider where the Provider's SE counterpart applet is already preselected.

(a) Constructor: Provider(Channel channel)

Encapsulates the defined channel by a Provider object that can be used for performing a service operation on it. This constructor has to be called by derived Provider classes during the instantiation.

Parameters:

channel - the channel that shall be used by this Provider for service operations.

Errors:

IllegalStateError - if the defined channel is closed.

(b) Method: Channel getChannel()

Returns the channel that is used by this provider.
This returned channel can also be used by other providers.

Return value:

The channel instance that is used by this provider.

7.5 Crypto API

The crypto API that is native to the mobile operating system shall be used.

For SE vendors, it guarantees that the crypto functionality offered by the SEs will be seen at the same level as others.

For application developers, they have to be careful to choose the crypto functionality provided by the SE when enumerating the available crypto providers.

For example, in a Java environment, it is recommended to use the Java Crypto Extension as the Java binding of the crypto API. More information on the JCE can be found in [7].
[7] serves two purposes:

- It contains information about the API side of the Java Cryptography Architecture, that may be used by developers to learn how to use this API.
- It contains information about the SPI (Service Provider Interface) side of the Java Cryptography Architecture, i.e. how to add new cryptography capabilities to a Java platform, by implementing a Provider, and how to declare and use the new providers.

In a native environment, where C/C++ is used as the programming language, it is recommended to use PKCS#11 as the native binding of the crypto API. More information on PKCS#11 can be found in [5].

In a mixed environment where Java and C/C++ are cohabiting, the PKCS#11 implementations should be made available to the Java applications. This is typically done by defining a JCE Provider for PKCS#11, that behaves as a Java binding for PKCS#11.

The following is an example of such a mixed architecture.
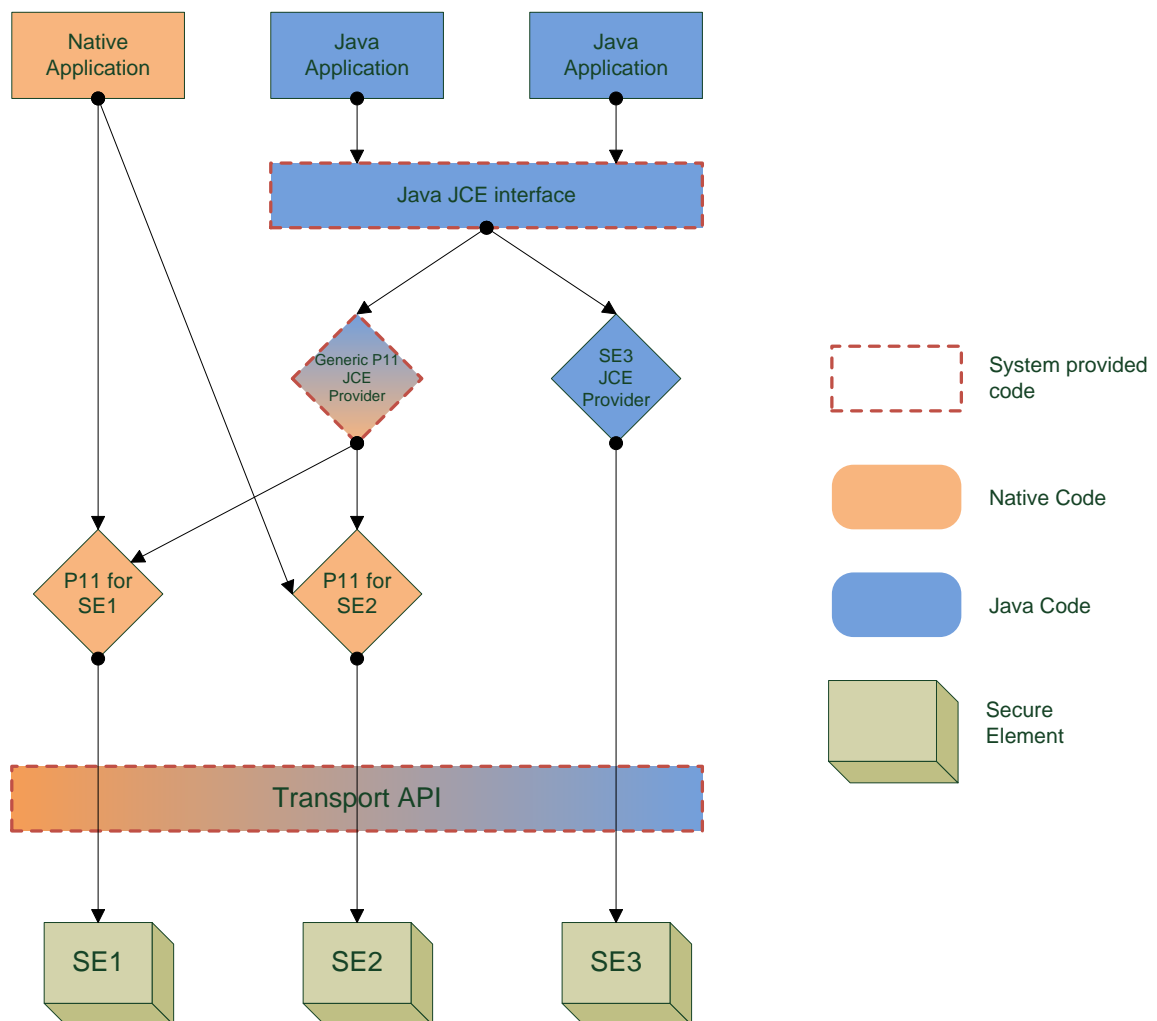
Figure 7-4 Crypto API architecture

7.5.1   Extensibility

A requirement of this API is to provide the functionality to add system wide crypto providers during runtime (without flashing the device) to support the different card implementations in the field.

For example, in a Java environment, the JCE provider architecture should be used to declare new providers (by SE vendors) and to lookup for JCE providers (by application developers).

In a native environment, a mechanism to declare new PKCS#11 implementation (typically as shared libraries) must be available (to be used by PKCS#11 implementers), and reciprocally a mechanism to choose between the available PKCS#11 libraries must be available (to be used by application developers).

7.5.2   Extending by Shared Libraries

On systems that support the use of shared libraries, this mechanism can be used to provide new implementations of crypto providers. For example, PKCS#11 implementations can be provided as shared libraries. The mechanism to register and discover these shared libraries is yet to be defined (it is not part of PKCS#11).

7.5.3   Extending by Applicative plugins

On systems where extensibility can only be achieved by installing new applications, and which provide a way to perform inter-application communication (e.g. IPC), a plugin mechanism can be used.

A generic provider based on the IPC capabilities can be offered by the system, to be used by the applications. The role of this generic provider is to lookup for plugin applications implementing a specific crypto service interface, list them to the crypto-aware applications, (enumerated as regular crypto providers) and when a crypto-aware application selects a particular provider, establish the connection to the plugin application actually implementing the crypto operations and forward all the requests thanks to the IPC mechanism.

The following is an illustration of such an implementation on a Java-based environment, using a JCE provider (the same could apply to a native-based environment, using a PKCS#11 library):
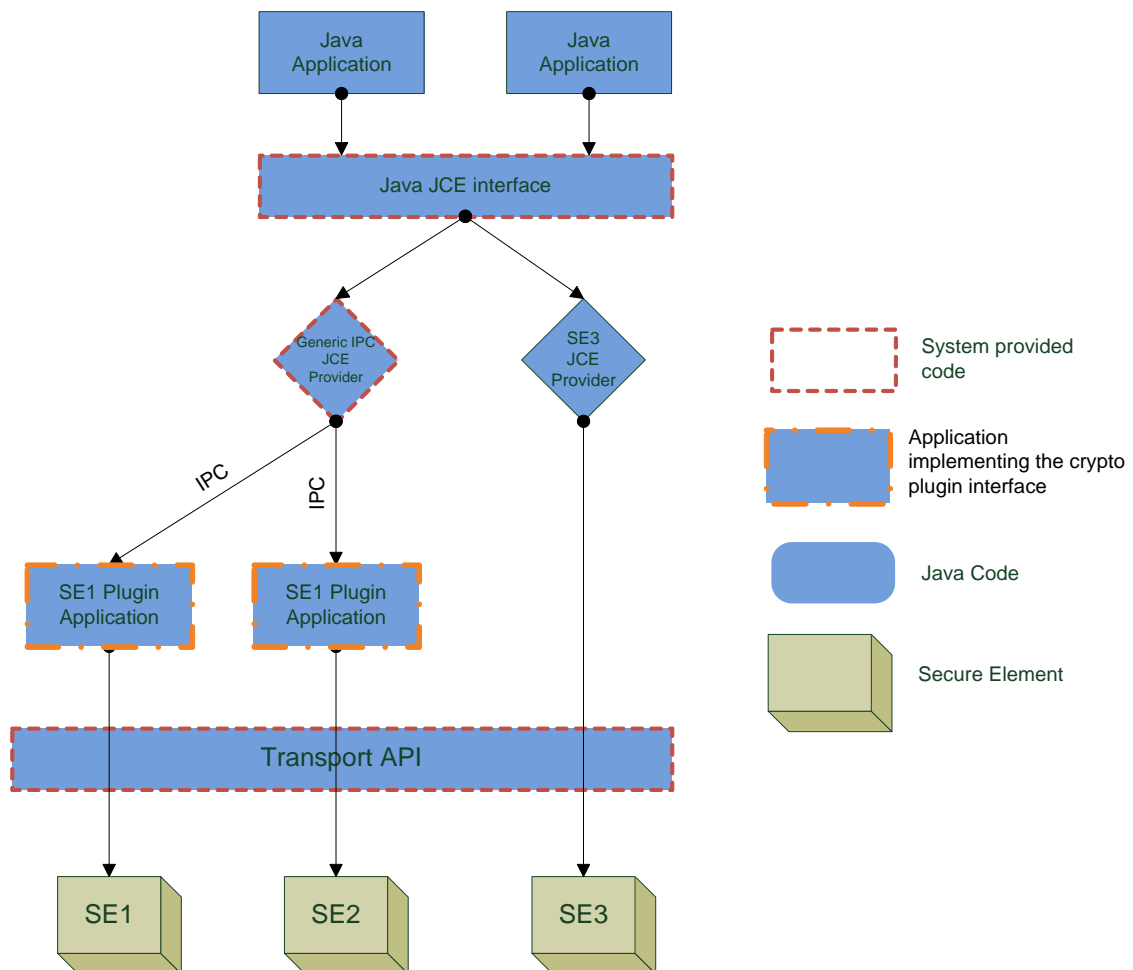
Figure 7-5: Crypto API architecture with plugin Applications

With this architecture, when support for a new type of SE providing crypto services has to be added to the system, it is just a matter of installing a new plugin application implementing the protocol to interact properly with this new type of crypto provider.

7.5.4   Integration with the Transport API

The Crypto API(s) should be implemented on top of the Transport API; if it is not, the system should behave as if it was. For example, if the Crypto API requires access to the basic channel of a SE, then it is subject to the same rules as the other applications: access to the basic channel is provided only if the application calling the Crypto API is authorised to access the basic channel, and if it is currently not locked. In the same way, if the basic channel is locked by the Crypto API, it is not available to other applications.

If the PKCS#11 libraries are implemented on top of a PC/SC layer or an equivalent layer that gives a reader-access level, then it is recommended that APDUs sent over this layer are filtered and processed to be translated into commands for the Transport API.

For example, if a native application sends a MANAGE_CHANNEL command and then a SELECT_BY_DF_NAME command, this should be translated into a call to openLogicalChannel.

7.6 Discovery API

This API provides means for the applications to lookup for a SE, based on a search criterion. The rationale behind such an API is to factorise this lookup code in a system-provided API, reducing the development cost of this part of each application.

This API relies on an object-oriented approach: the lookup method is shared, but the criterion is implemented as a separate class that can be derived. A set of basic criterion class is provided, they include:
- Search by ATR.
- Search by Historical bytes.
- Search by AID.

If these basic research criteria are not sufficient to fulfil a specific application's needs, then the application can provide its own criterion object. This object will have its isMatching() method called for each SE present in the system, and will be able to use the Transport API (or any other Service API) to implement a specific matching algorithm. Examples of such specific algorithm are:
- Analysis of EFdir file (if available).
- Analysis of GlobalPlatform Status information (if available).

d = SEDiscovery(<SE matching criteria>)

Reader = d.getFirstMatch()

Reader = d.getNextMatch()
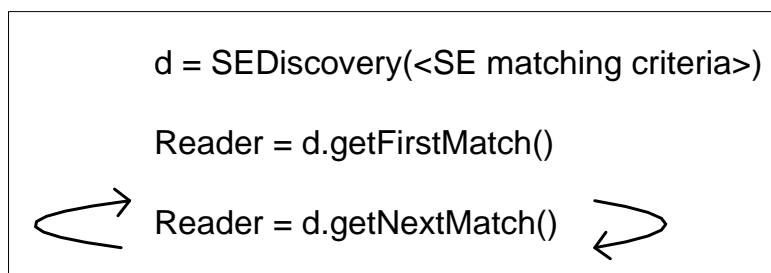
Figure 7-6: Discovery mechanism

7.6.1   Class: SEDiscovery

Instances of this class must be created by the applications to start a discovery process.

When created, they are configured with a SEService and an object that will perform the recognition algorithm.

> **(a) Constructor: SEDiscovery (SEService service, SERecognizer recognizer)**

Creates a discovery object that will perform a discovery algorithm specified by the recognizer object, and will be applied to the given SEService.

Parameters:

service - the SEService used to perform the discovery. Cannot be null.
recognizer - an SERecognizer instance, whose isMatching will be called. Cannot be null.

Errors:

IllegalParameterError – if one of the parameters is null.

> **(b) Method: Reader getFirstMatch()**

Returns the first SE reader containing a SE that matches the search criterion.
Actually starts a full discovery process:
– SE readers are enumerated
– For the first reader, if a SE is present, open a session.
– On this session, call the isMatching method of the SERecognizer object given at construction time.
– The session is closed.
– If the isMatching method returns false, the process is continued with the next reader.
– If the isMatching method returns true, the reader object is returned.

The sessions used by the discovery process are closed to avoid the risk of leaks: if they were opened and returned to the caller, there would be a risk for the caller to forget to close them.

Calling getFirstMatch twice simply restarts the discovery process (e.g. probably returns the same result, unless a SE has been removed).

Return Value:

The first matching SE reader, or null if there is none.

Errors:

NONE. All errors must be caught within the implementation.

> **(c) Method: Reader getNextMatch()**

Returns the next SE reader containing a SE that matches the search criterion.
Actually continues the discovery process:
– For the next reader in the enumeration, if a SE is present, open a session.
– On this session, call the isMatching method of the SERecognizer object given at construction time.
– The session is closed.
– If the isMatching method returns false, the process is continued with the next reader.
– If the isMatching method returns true, the reader object is returned.

Return Value:

The next matching SE reader, or null if there is none.

Errors:

IllegalStateError - if the getNextMatch() method is called without calling getFirstMatch() before, since the creation of the SEDiscovery object, or since the last call to getFirstMatch or getNextMatch that returned null.

7.6.2   Class: SERecognizer

Base class for recognizer classes.
Extended by system-provided recognizers, or by custom recognizers.

(a) Method: boolean isMatching(Session session)

This is a call-back method that will be called during the discovery process, once per SE inserted in a reader. Application developers can use the given session object to perform any discovery algorithm they think is appropriate. They can use the Transport API or any other API, conforming to access control rules & policy, like for regular application code (i.e. this is not privileged code).

Parameters:

session - a session object that is used to perform the discovery. Never null.

Return value:

A boolean indicating whether the SE to which the given session has been open is matching with the recognition criterion implemented by this method.

Errors:

NONE. All errors must be caught within the implementation of the method, and must be translated in a "false" result.

7.6.3   Class: SERecognizerByATR

Instances of this class can be used to find a SE with a specific ATR (or ATR pattern).

(a) Constructor: SERecognizerByATR (byte[] atr, byte[] mask)

Parameters:

atr - a byte array containing the ATR bytes values that are searched for.
mask - a byte array containing an AND-mask to be applied to the SE ATR values before to be compared with the searched value.

Errors:

IllegalParameterError – if ATR is invalid.

7.6.4   Class: SERecognizerByHistoricalBytes

Instances of this class can be used to find a SE with a specific value in their historical bytes.

(a) Constructor: SERecognizerByHistoricalBytes (byte[] values)

Parameters:

values - byte array, to be checked for presence in the historical bytes.

Errors:

IllegalParameterError – if historical bytes are invalid.

7.6.5   Class: SERecognizerByAID

Instances of this class can be used to find a SE implementing a specific applet, identified by its AID. The presence of such an applet is verified by trying to open a channel to this applet. The opened channel, if any, is closed before the end of the isMatching method.

> **(a) Constructor: SERecognizerByAID (byte[] aid)**

Parameters:
aid - byte array holding the AID to be checked for presence.

Errors:
IllegalParameterError – if AID is invalid.

7.7 File management

API for file management to read and write the content of files in an ISO/IEC 7816-4 compliant file system provided by the SE's OS or applet (installed in the SE).
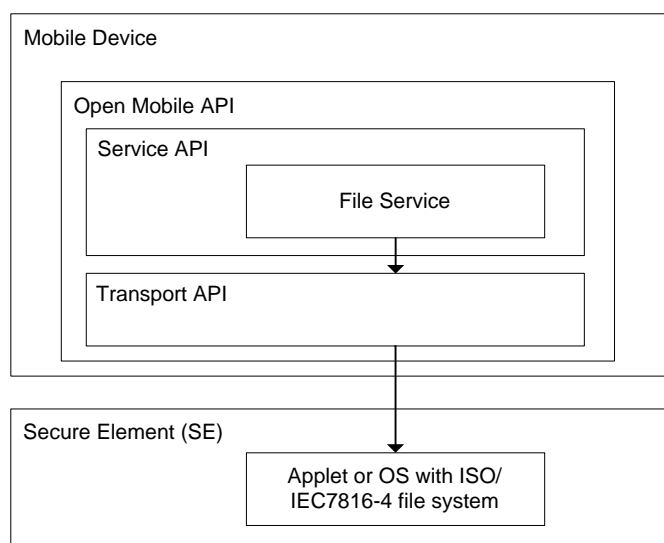
Figure 7-7: File management overview

7.7.1   Class: FileViewProvider

This Provider class simplifies file operations on SEs with a file structure specified in ISO/IEC 7816-4.

Methods are provided that allows file content to be read or written. If the read or write operation is not allowed because security conditions are not satisfied, a SecurityError will be returned. It must be considered that a file operation can only be applied onto a file which has a corresponding structure.

Prerequisites:
This Provider requires an ISO/IEC 7816-4 compliant file system on the SE. If this file system is implemented by an applet within the SE then this applet must be preselected before this Provider can be used, in case that the applet is not default selected (e.g. the GSM Applet as default selected applet on a UICC).

Notes:
- If used by multiple threads, synchronisation is up to the application.
- Each operation needs an access to the SE. If access cannot be granted because of a closed channel or a missing security condition, the called method will return an error.

simalliance

- Using the basic channel for accessing the file system of the UICC (provided by the default selected GSM Applet) implies the risk of interferences from the baseband controller, as the baseband controller works internally on the basic channel and can modify the current file selected state on the basic channel anytime. This means a file selection performed by this FileViewProvider does not guarantee a permanent file selection state on the UICC's basic channel and the application using the FileViewProvider has to take care to have the needed file selection state. The FileViewProvider itself cannot avoid interferences from the baseband controller on the basic channel, but the risk could be minimised if the application using the FileViewProvider performs implicit selections for the file operation or performs the file selection immediately before the file operation.

(a) Constant: CURRENT_FILE

Indicates for file operation methods that the currently selected file shall be used for the file operation.

(b) Constant: INFO_NOT_AVAILABLE

Indicates that the demanded information is not available.

(c) Constructor: FileViewProvider(Channel channel)

Encapsulates the defined channel by a FileViewProvider object that can be used for performing file operations on it.

Parameters:

channel - the channel that shall be used by this Provider for file operations.

Errors:

IllegalStateError - if the defined channel is closed.

Note:

A file must be selected before a file operation can be performed. The file can be implicitly selected via a short file identifier (SFI), by the file operation method itself or explicitly by defining the file ID (FID) with selectByFID(int) or path with selectByPath(String, boolean).

(d) Method: FCP selectByPath(String path, boolean fromCurrentDF)

Selects the file specified by a path.

The path references the SE file by a path (concatenation of file IDs and the order of the file IDs is always in the direction 'parent to child') in the following notation: "DF1:DF2:EF1". e.g. "0023:0034:0043". The defined path is applied to the SE as specified in ISO/IEC 7816-4. Note: For performing read or write operations on a file the last knot in the path must reference an EF that can be read or written.

Parameters:

path - the path that references a file (DF or EF) on the SE. This path shall not contain the current DF or MF at the beginning of the path.
fromCurrentDF - if true then the path is selected from the current DF,
if false then the path is selected from the MF.

Return value:

The FCP containing information about the selected file.

Errors:

IllegalParameterError - if the defined path is invalid.
IllegalStateError - if the defined channel is closed.

SecurityError - if the operation is not allowed because the security conditions are not satisfied.
IllegalReferenceError - if the file could not be selected.
OperationNotSupportedError - if this operation is not supported.

Notes:
- A file must be selected before a file operation can be performed.
- This method is based on the ISO/IEC 7816-4 command SELECT.

(e) Method: FCP selectByFID(int fileID)
Selects the file specified by the FID.

The file ID references the SE file (DF or EF) by a FID. The FID consists of a two byte value as defined in ISO/IEC 7816-4.

Parameters:
fileID - the FID that references the file (DF or EF) on the SE. The FID must be in the range of (0x0000-0xFFFF).

Return value:
The FCP containing information about the selected file.

Errors:
IllegalParameterError - if the defined fileID is not valid.
IllegalStateError - if the defined channel is closed.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
IllegalReferenceError - if the File could not be selected.
OperationNotSupportedError - if this operation is not supported.

Notes:
- A file must be selected before a file operation can be performed.
- This method is based on the ISO/IEC 7816-4 command SELECT.

(f) Method: FCP selectParent()
Selects the parent DF of the current DF.

The parent DF of the currently selected file is selected according to ISO/IEC 7816-4. If the currently selected file has no parent then nothing will be done.

Return value:
The FCP containing information about the selected file.

Errors:
IllegalStateError - if the defined channel is closed.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
IllegalReferenceError - if the File could not be selected.
OperationNotSupportedError - if this operation is not supported.

Notes:
- A file must be selected before a file operation can be performed.
- This method is based on the ISO/IEC 7816-4 command SELECT.

(g) Method: Record readRecord(int sfi, int recID)

Returns the record which corresponds to the specified record ID. If the record is not found, then null will be returned.

Parameters:

sfi - the SFI of the file which shall be selected for this read operation. CURRENT_FILE can be applied if the file is already selected. The sfi must be in the range of (1-30).
recID - the record ID that references the record that should be read.

Return value:

The record which corresponds to the specified record ID.

Errors:

IllegalStateError - if the used channel is closed.
IllegalStateError - if no file is currently selected.
IllegalStateError - if the currently selected file is not a record based file.
IllegalStateError - if the record could not not be read.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
IllegalReferenceError - if the file could not be selected via SFI.
IllegalParameterError - if the defined SFI is not valid.
IllegalParameterError - if the defined record ID is invalid.
OperationNotSupportedError - if this operation is not supported.

Note:

This method is based on the ISO/IEC 7816-4 command READ RECORD.

(h) Method: void writeRecord(int sfi, Record rec)

Writes a record into the specified file.

Parameters:

sfi - the SFI of the file which shall be selected for this write operation. CURRENT_FILE can be applied if the file is already selected. The sfi must be in the range of (1-30).
rec - the Record that shall be written.

Errors:

IllegalStateError - if the used channel is closed.
IllegalStateError - if no file is currently selected.
IllegalStateError - if the currently selected file is not a record based file.
IllegalStateError - if the record could not be written.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
IllegalReferenceError - if the file could not be selected via SFI.
IllegalParameterError - if the defined record is invalid.
IllegalParameterError - if the defined SFI is not valid.
OperationNotSupportedError - if this operation is not supported.

Note:

This method is based on the ISO/IEC 7816-4 command APPEND RECORD and UPDATE RECORD (which replaces existing bytes).

(i) Method: int[] searchRecord(int sfi, byte[] searchPattern)

Returns the record numbers that contains the defined search pattern.

Parameters:
sfi - the SFI of the file which shall be selected for this search operation. CURRENT_FILE can be applied if the file is already selected. The sfi must be in the range of (1-30).
searchPattern - the pattern that shall match with records.

Return value:
A list of record numbers (position 1..n of the record in the file) of the records which match to the search pattern. If no record matches then null will be returned.

Errors:
IllegalParameterError - if the defined SFI is not valid.
IllegalStateError - if the used channel is closed.
IllegalStateError - if no file is currently selected.
IllegalStateError - if the currently selected file is not a record based file.
IllegalStateError - if the search pattern is empty.
IllegalStateError - if the search pattern is too long.
IllegalStateError - if the data could not be searched.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
IllegalReferenceError - if the file could not be selected via SFI.
OperationNotSupportedError - if this operation is not supported.

Note
This method is based on the ISO/IEC 7816-4 command SEARCH RECORD with simple search.

(j) Method: byte[] readBinary(int sfi, int offset, int length)
Reads content of the selected transparent file at the position specified by offset and length.

Parameters:
sfi - the SFI of the file which shall be selected for this read operation. CURRENT_FILE can be applied if the file is already selected. The sfi must be in the range of (1-30).
offset - defines the start point of the file where the data should be read.
length - defines the length of the data which should be read.

Return value:
The data read from the file or null if no content is available.

Errors:
IllegalParameterError - if the defined SFI is not valid.
IllegalParameterError - if the defined offset and length could not be applied.
IllegalStateError - if the used channel is closed.
IllegalStateError - if no file is currently selected.
IllegalStateError - if the currently selected file is not a transparent file.
IllegalStateError - if the data could not be read.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
IllegalReferenceError - if the file could not be selected via SFI.
OperationNotSupportedError - if this operation is not supported.

simalliance

Note:
This method is based on the ISO/IEC 7816-4 command READ BINARY.

(k) Method: void writeBinary(int sfi, byte[] data, int offset, int length)
Writes the defined data into the selected file at the position specified by offset and length.

Parameters:
sfi - the SFI of the file which shall be selected for this write operation. CURRENT_FILE can be applied if the file is already selected. The sfi must be in the range of (1-30).
data - the data which shall be written.
offset – defines the position in the file where the data should be stored.
length - defines the length of the data which shall be written.

Errors:
IllegalParameterError - if the defined SFI is not valid.
IllegalParameterError - if the defined data array is empty or too short.
IllegalParameterError - if the defined offset and length could not be applied.
IllegalStateError - if the used channel is closed.
IllegalStateError - if no file is currently selected.
IllegalStateError - if the currently selected file is not a transparent file.
IllegalStateError - if the data could not be written.
SecurityError - if the operation is not allowed because the security conditions are not satisfied.
IllegalReferenceError - if the file could not be selected via SFI.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command UPDATE BINARY.

7.7.2   Class: FileViewProvider:FCP
File control parameter contains information of a selected file. FCPs are returned after a file select operation. This class is based on the ISO/IEC 7816-4 FCP returned by the SELECT command as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects).

(a) Method: byte[] getFCP()
Returns the complete FCP as byte array.

Return value:
The complete FCP as byte array.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (File control parameter data objects).

(b) Method: int getFileSize()
Returns the file size of the selected file (number of data bytes in the file, excluding structural information).

Return value:
The file size depending on the file type:
− Transparent EF: the length of the body part of the EF.

– Linear fixed or cyclic EF: record length multiplied by the number of records of the EF.

INFO_NOT_AVAILABLE if the information is not available.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (File control information) in table 12 (file control parameter data objects).

(c) Method: int getTotalFileSize()

Returns the total file size of the selected file (number of data bytes in the file, including structural information if any).

Return value:
The total file size depending on the file type:
– DF/MF: the total file size represents the sum of the file sizes of all the EFs and DFs contained in this DF, plus the amount of available memory in this DF. The size of the structural information of the selected DF itself is not included.
– EF: the total file size represents the allocated memory for the content and the structural information (if any) of this EF.

INFO_NOT_AVAILABLE if the information is not available.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (file control information) in table 12 (file control parameter data objects).

(d) Method: int getFID()

Returns the file identifier of the selected file.

Return value:
The file identifier of the selected file.
INFO_NOT_AVAILABLE if the FID of the selected file is not available.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (file control information) in table 12 (file control parameter data objects).

(e) Method: int getSFI()

Returns the short file identifier of the selected EF file.

Return value:
The short file identifier of the selected file.
INFO_NOT_AVAILABLE if selected file is not an EF or an SFI is not available.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (file control information) in table 12 (file control parameter data objects).

(f) Method: int getMaxRecordSize()

Returns the maximum record size in case of a record based EF.

Return value:
The maximum record size in case of a record based EF.

INFO_NOT_AVAILABLE if the currently selected file is not record based or the information cannot be fetched.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (file control information) in table 12 (file control parameter data objects).

(g) Method: int getNumberOfRecords()
Returns the number of records stored in the EF in case of a record based EF.

Return value:
The number of records stored in the EF in case of a record based EF.
INFO_NOT_AVAILABLE if the currently selected file is not record based or the information cannot be fetched.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (file control information) in table 12 (file control parameter data objects).

(h) Method: int getFileType()
Returns the file type of the currently selected file.

Return value:
The file type:
– (0) DF
– (1) EF
INFO_NOT_AVAILABLE if the information cannot be fetched.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (file control information) in table 12 (file control parameter data objects). The file type is based on the definition in table 14 (file descriptor byte).

(i) Method: int getFileStructure()
Returns the structure type of the selected EF.

Return value:
The structure type of the selected file:
– (0) NO_EF
– (1) TRANSPARENT
– (2) LINEAR_FIXED
– (3) LINEAR_VARIABLE
– (4) CYCLIC
INFO_NOT_AVAILABLE if the information cannot be fetched.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (file control information) in table 12 (file control parameter data objects). The file structure is based on the definition in table 14 (file descriptor byte).

(j) Method: int getLCS()
Returns the lifecycle state of the currently selected file.

Return value:
The lifecycle state:
– (0) NO_INFORMATION_GIVEN
– (1) CREATION_STATE
– (2) INITIALISATION_STATE
– (3) OPERATIONAL_STATE_ACTIVATED
– (4) OPERATIONAL_STATE_DEACTIVATED
– (5) TERMINATION_STATE
INFO_NOT_AVAILABLE if the information is not available.

Note:
This method is based on the FCP control parameter as specified in ISO/IEC 7816-4 in chapter 5.3.3 (file control information) in table 12 (file control parameter data objects). The lifecycle status is based on the definition in table 13 (lifecycle status byte).

7.7.3   Class: FileViewProvider:Record

Record class serves as a container for record data. The created record (as immutable object) can be used to read record data from a file or to write record data to a file.

(a) Contructor: Record(int id, byte[] data)
Creates a record instance which can be used to store record data.

Parameter:
id - the record id that shall be stored.
data - the data that shall be stored.

(b) Method: int getID()
Returns the record ID of this record.

Return value:
The record ID of this record.

(c) Method: byte[] getData()
Returns the data of this record.

Return value:
The data of this record.

7.8　Authentication service

Provides an API to perform a PIN authentication on the SE to enable an authentication state.

Use cases:

- Performing an operation on the SE which requires a user authentication.
  - o　e.g. for reading a file from the SE's file system which is PIN protected.
  - o　e.g. for using a key from the SE which requires a PIN authentication.
- Moreover the API allows the management of SE PINs with management commands like reset, change, activate and deactivate.

```
Mobile Device
  ┌─────────────────────────────────────┐
  │ Open Mobile API                     │
  │  ┌───────────────────────────────┐  │
  │  │ Service API                   │  │
  │  │        ┌────────────────────┐ │  │
  │  │        │ Authentication     │ │  │
  │  │        │ Service            │ │  │
  │  │        └────────────────────┘ │  │
  │  └───────────────────────────────┘  │
  │  ┌───────────────────────────────┐  │
  │  │ Transport API                 │  │
  │  └───────────────────────────────┘  │
  └─────────────────────────────────────┘

Secure Element (SE)
        ┌────────────────────┐
        │ Applet or OS with  │
        │ ISO/IEC7816-4 PIN  │
        │ commands           │
        └────────────────────┘
```
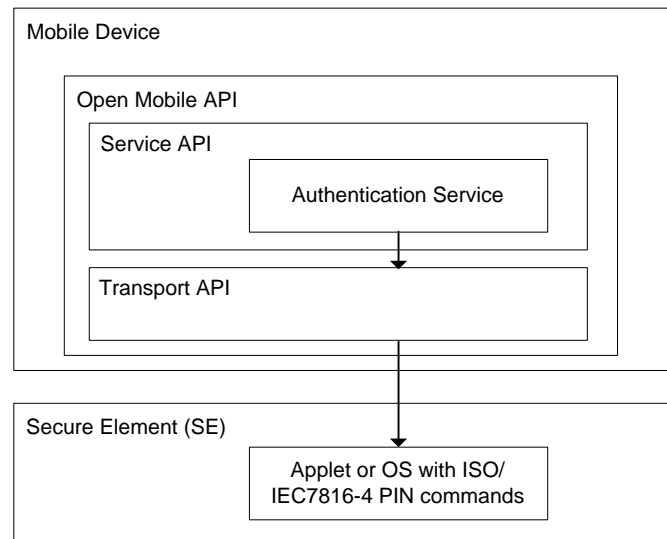
Figure 7-8: Authentication service overview

7.8.1　　　　Class: AuthenticationProvider

This Authentication class can be used to privilege a certain communication channel to the SE for operations that require a PIN authentication. Besides the PIN verification for authentication, this class also provides PIN management commands for changing, deactivating or activating PINs.

Prerequisites:

The PIN operations performed by this AuthenticationProvider class are based on the ISO/IEC 7816-4 specification and require a preselected applet on the specified communication channel to the SE that implements ISO/IEC 7816-4 compliant PIN commands.

Notes:

- If used by multiple threads, synchronisation is up to the application.
- Each operation needs access to the SE. If access cannot be granted because of a closed channel or a missing security condition, the called method will return an error.

(a)　Constructor: AuthenticationProvider(Channel channel)

Encapsulates the defined channel by an AuthenticationProvider object that can be used for applying PIN commands on it.

Parameters:

channel - the channel that should be privileged for operations which require a PIN authentication.

Errors:
IllegalStateError - if the defined channel is closed.

(b) Method: boolean verifyPin(PinID pinID, byte[] pin)
Performs a PIN verification.

Parameters:
pinID - the PIN ID references the PIN in the SE which shall be used for the verification.
pin - the PIN that shall be verified.

Return value:
True if the authentication was successful.
False if the authentication fails.

Errors:
IllegalParameterError - if the PIN value has bad coding or a wrong length (empty or too long).
IllegalStateError - if the used channel is closed.
IllegalReferenceError - if the PIN reference as defined could not be found in the SE.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command VERIFY.
When the PIN is blocked, the method returns false. Clients have to use getRetryCounter to check for a blocked PIN.

(c) Method: void changePin(PinID pinID, byte[] oldPin, byte[] newPin)
Changes the PIN.

Parameters:
pinID - the PIN ID references the PIN in the SE which shall be changed.
oldPin - the old PIN that shall be changed.
newPin - the PIN that shall be set as the new PIN.

Errors:
IllegalParameterError - if the value of oldPin or newPIN has bad coding or a wrong length (empty or too long).
IllegalStateError - if the used channel is closed.
SecurityError - if old PIN does not match with the PIN stored in the SE. The PIN is not changed.
IllegalReferenceError - if the PIN reference as defined could not be found in the SE.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command CHANGE REFERENCE DATA.

(d) Method: void resetPin(PinID pinID, byte[] resetPin, byte[] newPin)
Resets the PIN with the reset PIN or just resets the retry counter.

Parameters:
pinID - the PIN ID references the PIN in the SE which shall be reset.
resetPin - the reset PIN that shall be used for reset.

newPin - the PIN that shall be set as new PIN. Can be omitted with null if just the reset counter shall be reset.

Errors:
IllegalParameterError - if the value of resetPin or newPIN has bad coding or a wrong length (empty or too long).
IllegalStateError - if the used channel is closed.
SecurityError - if resetPIN does not match with the "reset PIN" stored in the SE. The PIN or reset counter is not changed.
IllegalReferenceError - if the PIN ID reference as defined could not be found in the SE.
OperationNotSupportedError - if this operation is not supported (e.g. PIN is not defined).

Note:
This method is based on the ISO/IEC 7816-4 command RESET RETRY COUNTER.

(e) Method: int getRetryCounter(PinID pinID)
Returns the retry counter of the referenced PIN.

Parameters:
pinID - the PIN ID references the PIN in the SE and its retry counter.

Return value:
The retry counter of the referenced PIN.

Errors:
IllegalStateError - if the used channel is closed.
IllegalReferenceError - if the PIN reference as defined could not be found in the SE.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command VERIFY.

(f) Method: void activatePin(PinID pinID, byte[] pin)
Activates the PIN. Thus a deactivated PIN can be used again.

Parameters:
pinID - the PIN ID references the PIN in the SE which shall be activated.
pin - the verification PIN for activating the PIN if required. Can be omitted with null if not required.

Errors:
IllegalParameterError - if the PIN value has bad coding or a wrong length (empty or too long).
IllegalStateError - if the used channel is closed.
SecurityError - if the defined pin does not match with the PIN needed for the activation. The PIN state will not be changed.
IllegalReferenceError - if the PIN reference as defined could not be found in the SE.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command ENABLE VERIFICATION REQUIREMENT.

(g) Method: void deactivatePin(PinID pinID, byte[] pin)

Deactivates the PIN. Thus the objects which are protected by the PIN can now be used without this restriction until activatePin() is called.

Parameters:
pinID - the PIN ID references the PIN in the SE which shall be deactivated.
pin - the verification PIN for deactivating the pin if required. Can be omitted with null if not required.

Errors:
IllegalParameterError - if the PIN value has bad coding or a wrong length (empty or too long).
IllegalStateError - if the used channel is closed.
SecurityError - if the defined PIN does not match with the PIN needed for the deactivation. The PIN state will not be changed.
IllegalReferenceError - if the PIN reference as defined could not be found in the SE.
OperationNotSupportedError - if this operation is not supported.

Note:
This method is based on the ISO/IEC 7816-4 command DISABLE VERIFICATION REQUIREMENT.

7.8.2   Class: AuthenticationProvider:PinID

This PIN ID uniquely identifies a PIN in the SE system. The PIN ID is defined as specified in ISO/IEC 7816-4 and can be used to reference a PIN in an ISO/IEC 7816-4 compliant system.

(a) Constructor: PinID(int id, boolean local)

Creates a PIN ID (reference) to identify a PIN within a SE. The created PIN ID (as immutable object) can be specified on all PIN operation methods provided by the AuthenticationProvider class.

Parameters:
id - the ID of the PIN (value from 0x00 to 0x1F).
local - defines the scope (global or local). True if the PIN is local. Otherwise, false.

Errors:
IllegalParameterError - if the defined ID is invalid.

Note:
This constructor is based on the P2 reference data for PIN related commands as specified in ISO/IEC 7816-4 in chapter 7.5 (basic security handling). Local set to true indicates specific reference data and local set to false indicates global reference data according to ISO/IEC 7816-4. The ID indicates the number of the reference data (qualifier) according to ISO/IEC 7816-4.

(b) Method: int getID()

Returns the PIN ID.

Return value:
The PIN ID.

Note:
This method is based on the P2 reference data for PIN related commands as specified in ISO/IEC 7816-4 in chapter 7.5 (basic security handling). The ID indicates the number of the reference data (qualifier) according to ISO/IEC 7816-4.

> *(c) Method: boolean isLocal()*

Identifies if the PIN is local or global.

Return value:
True if the PIN is local. Otherwise, false.

Note:
This method is based on the P2 reference data for PIN related commands as specified in ISO/IEC 7816-4 in chapter 7.5 (basic security handling). Local set to true indicates specific reference data and local set to false indicates global reference data according to ISO/IEC 7816-4.

7.9　PKCS#15 API

The PKCS#15 standard is a structured way to store and organise data. The PKCS#15 service API simplifies access to PKCS#15 file systems according to v1.1 of the PKCS#15 specification. Classes and methods are provided to retrieve the elementary PKCS#15 data structures, such as ODF (Object Directory File) and TokenInfo.

PKCS#15 file systems can be used to store cryptographic data, but also, any kind of data using application-specific files and OIDs. For example in the OMA-DM use case, the bootstrap data can be stored in the SE's file system using a dedicated PKCS#15 file structure. This PKCS#15 API can be used by an OMA-DM client application to retrieve the bootstrap data from a SE.

The preferred way to select a PKCS#15 file system is through the PKCS#15 AID (A0 00 00 00 63 50 4B 43 53 2D 31 35), however, a legacy file system can reference a PKCS#15 data structure though the EF(DIR).

Example of a typical PKCS#15 file system:
```
ADF(PKCS#15)        : AID = A0 00 00 00 63 50 4B 43 53 2D 31 35
 |-EF(ODF)          : FID = 5031
 |-EF(TokenInfo)    : FID = 5032
 |-EF(PrKDF)        : optional, referenced by EF(ODF)
 |-EF(PuKDF)        : optional, referenced by EF(ODF)
 |-EF(CDF)          : optional, referenced by EF(ODF)
 |-EF(DODF)         : optional, referenced by EF(ODF)
 |-EF(AODF)         : optional, referenced by EF(ODF)
```

Example of a legacy file system with a PKCS#15 structure:
```
MF                    : FID=3F00
 |-EF(DIR)            : FID=2F00
 |-DF(PKCS#15)        : referenced by EF(DIR)
    |-EF(ODF)         : FID = 5031
    |-EF(TokenInfo) : FID = 5032
```
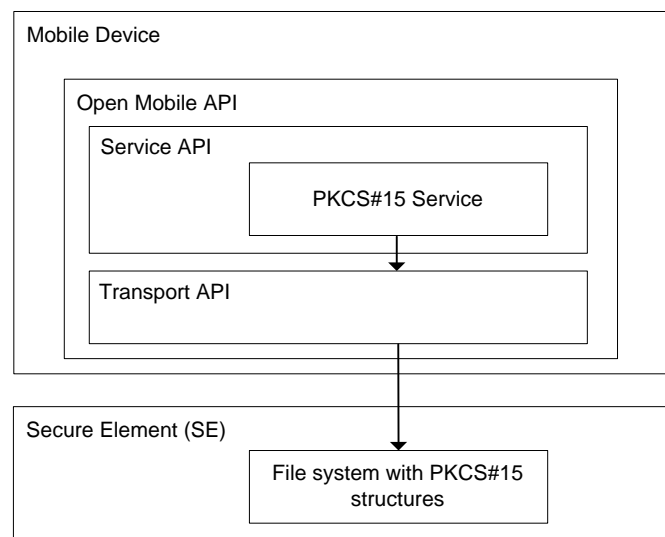

Figure 7-9: PKCS#15 service overview

7.9.1   Class: PKCS15Provider

This Provider class offers basic services to access a PKCS#15 file system. This Provider requires a PKCS#15 data structure on the SE and a channel instance allowing access to this PKCS#15 data structure.

(a) Constant: byte[] AID_PKCS15

Default PKCS#15 AID (A0 00 00 00 63 50 4B 43 53 2D 31 35).

(b) Constructor: PKCS15Provider(Channel channel)

Encapsulates the defined channel by a PKCS#15 file system object. This method checks the presence of the EF(ODF) (Object Directory File) with file identifier 5031 and of the EF(TokenInfo) with file identifier 5032. Both files are mandatory and must be present in a valid PKCS#15 file system.

This method must first try to select EF(ODF) and EF(TokenInfo) on the provided channel. If the select fails, this method must try to locate a DF(PKCS#15) in the legacy file system using the EF(DIR) according to the data structure described in chapter 5.4 of the PKCS#15 specification (v1.1).

Parameters:

channel - the channel that shall be used by this Provider for file operations.

Errors:

IOError - if no PKCS#15 file system is detected on the provided channel.

(c) Method: byte[] getODF()

Returns the raw content of the EF(ODF) (Object Directory File).

Return value:

The EF(ODF) as a byte array. Must not be null.

(d) Method: byte[] getTokenInfo()

Returns the raw content of the EF(TokenInfo).

Return value:

The EF(TokenInfo) as a byte array. Must not be null.

(e) Method: Path[] getPrivateKeyPaths()

Returns an array of EF(PrKDF) paths (Private Key Directory Files). The PKCS#15 file system may contain zero, one or several EF(PrKDF).

Return value:

The array of EF(PrKDF) paths. May be null if empty.

(f) Method: Path[] getPublicKeyPaths()

Returns an array of EF(PuKDF) paths (Public Key Directory Files). The PKCS#15 file system may contain zero, one or several EF(PuKDF).

Return value:

The array of EF(PuKDF) paths. May be null if empty.

(g) Method: Path[] getCertificatePaths()

Returns an array of EF(CDF) paths (Certificate Directory Files). The PKCS#15 file system may contain zero, one or several EF(CDF).

Return value:

The array of EF(CDF) paths. May be null if empty.

(h) Method: Path[] getDataObjPaths()

Returns an array of EF(DODF) paths (Data Object Directory Files). The PKCS#15 file system may contain zero, one or several EF(DODF).

Return value:

The array of EF(DODF) paths. May be null if empty.

(i) Method: Path[] getAuthObjPaths()

Returns an array of EF(AODF) paths (Authentication Object Directory Files). The PKCS#15 file system may contain zero, one or several EF(AODF).

Return value:

The array of EF(AODF) paths. May be null if empty.

(j) Method: byte[] readFile(Path path)

Selects and reads a PKCS#15 file. The file may be a transparent or linear fixed EF. The 'index' and 'length' fields of the path instance will be used according to chapter 6.1.5 of the PKCS#15 specification (v1.1). In case of transparent EF, 'index' is the start offset in the file and 'length' is the length to read. In case of linear fixed EF, 'index' is the record to read.

Parameters:

path - path of the file.

Return value:

The file content as a byte array. Or null if the referenced path does not exist.

Errors:

SecurityError - if the operation cannot be performed, if a security condition is not satisfied.
OperationNotSupportedError - if this operation is not supported.
IOError - if the PKCS#15 file cannot be selected or read.

(k) Method: byte[] searchOID(byte[] dodf, String oid)

Parses the raw content of an EF(DODF) and searches for a specific OID Data Object. This method is a convenience method to simplify the access to OID Data Objects by applications, as described in chapter 6.7.4 of the PKCS#15 specification (v1.1). In many cases, the EF(DODF) contains a simple OID Data Object with a Path object, in order to reference an application-specific EF. For example, the OMA-DM specification requires a EF(DODF) containing the OID 2.23.43.7.1, followed by a path object, referencing the EF(DM_Bootstrap).

Parameters:

dodf - the raw content of an EF(DODF) to parse.
oid - the searched OID value (e.g. OMA-DM bootstrap OID is 2.23.43.7.1).

Return value:
The raw object value if OID has been found, null if not found.

Errors:
IllegalParameterError - if the OID is not correct.
OperationNotSupportedError - if this operation is not supported.

(l)   Method: Path decodePath(byte[] der)
Builds a path object using a DER-encoded (see ITU X.690 for DER-Coding) buffer.

Parameters:
der - the DER-encoded path object as a byte array.

Return value:
The path object.

Errors:
IllegalParameterError - if the defined path is not a correctly DER-encoded buffer.
OperationNotSupportedError - if this operation is not supported.

7.9.2   Class: PKCS15Provider:Path
This class represents a path object as defined in chapter 6.1.5 of the PKCS#15 specification (v1.1).

(a)   Constructor: Path(byte[] path)
Builds a path object without index and length (the path can be absolute as well as relative).

Parameters:
path - the path as a byte array.

Errors:
IllegalParameterError - if the path is not correct.

(b)   Constructor: Path(byte[] path, int index, int length)
Builds a path object with index and length (the path can be absolute as well as relative).

Parameters:
path - the path as a byte array.
Index - the index value.
length - the length value.

Errors:
IllegalParameterError - if the path, index or length is not correct.

(c)   Method: byte[] getPath()
Returns the path field of this path object.

Return value:
The path field.

(d)   Method: boolean hasIndexLength()
Checks whether this path object has an index and length fields.

Return value:

True if the index and length field is present, false otherwise.

(e) Method: int getIndex()

Returns the index field of this path object. The value of this field is undefined if the method hasIndexLength() returns false.

Return value:

The index field.

(f) Method: int getLength()

Returns the length field of this path object. The value of this field is undefined if the method hasIndexLength() returns false.

Return value:

The length field.

(g) Method: byte[] encode()

Encodes this path object according to DER (see ITU X.690 for DER-Coding).

Return value:

This path object as a DER-encoded byte array.

7.10 Secure Storage

The Secure Storage (SS) Service can be used to store and retrieve sensitive data on the SE. This API requires a SS Applet on the SE with an APDU interface as defined below.

Data is stored in a dictionary format (string, value). It is simpler to store data in the SS than with a PKCS#15Provider or FileViewProvider, which can, in principle, also be used to store data securely but in a more elaborate way.
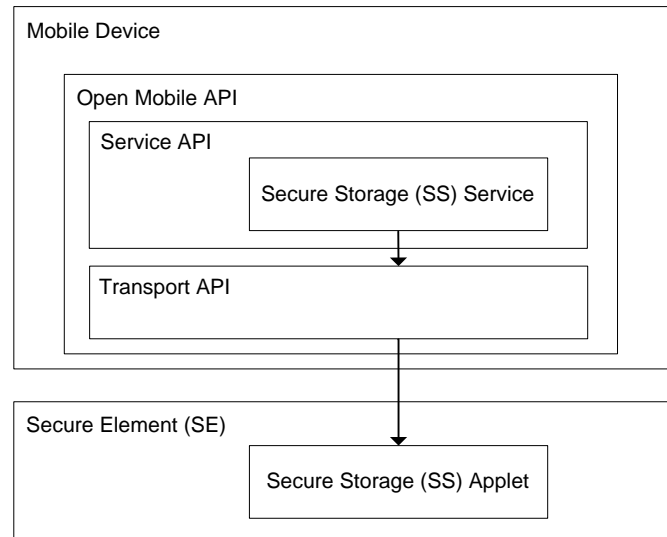
Figure 7-10: Secure Storage service overview

7.10.1　　　Class: SecureStorageProvider

This class provides an API to store and retrieve data on the SE which is protected in a secure environment. A default set of functionality that is always provided on every platform enables application developers to rely on this interface for secure data storage (e.g. credit card numbers, private phone numbers, passwords etc.). The interface should encapsulate any SE specifics as it is intended for device application developers who might not be familiar with SE or APDU internals.

Security Notes:

A PIN verification is required to grant access to the SS Applet where the Authentication Provider can be reused for the PIN operations. The SS Applet must separate the PIN verification on all logical channels to ensure that each device application needs to verify the PIN individually.

Prerequisites:

The SS operations performed by this Provider class are based on a SS located in the SE. The SS is usually realised by an applet (providing the defined SS APDU interface) that must be preselected on the specified communication channel to the SE before this Provider can be used.

Notes:

- If used by multiple threads, synchronisation is up to the application.
- Each operation needs access to the SE. If access cannot be granted because of a closed channel or a missing security condition, the called method will return an error.

(a) Constructor: SecureStorageProvider(Channel channel)

Creates a SecureStorageProvider instance which will be connected to the preselected SE SS Applet on a defined channel.

Parameters:

channel - the channel that shall be used by this Provider for operations on the SS.

Errors:

IllegalStateError - if the defined channel is closed.

(b) Method: void create(String title, byte[] data)

This command creates a SS entry with the defined title and data. The data can contain an uninterpreted byte stream of an undefined max length (e.g. names, numbers, image, media data, etc).

Parameters:

title - the title of the entry that shall be written. The maximum title length is 60. All characters must be supported by UTF-8.

data - the data of the entry that shall be written. If data is empty or null then only the defined title will be assigned to the new entry.

Errors:

IllegalParameterError - if the title already exists. All entry titles must be unique within the SS.

IllegalParameterError - if the title is incorrect: bad encoding or wrong length (empty or too long).

IllegalParameterError - if the data chain is too long.

IllegalStateError - if the used channel is closed.

SecurityError - if the PIN to access the SS Applet was not verified.

IOError – if the entry could not be created because of an incomplete write procedure.

(c) Method: void update(String title, byte[] data)

This command updates the data of the SS entry referenced by the defined title. The data can contain an uninterpreted byte stream of an undefined max length (e.g. names, numbers, image, media data, etc.).

Parameters:

title - the title of the entry that must already exist. The maximum title length is 60. All characters must be supported by UTF-8.

data - the data of the entry that shall be written. If data is empty or null then the data of the existing entry (referenced by the title) will be deleted.

Errors:

IllegalParameterError - if the title does not already exist.

IllegalParameterError - if the title is incorrect: bad encoding or wrong length (empty or too long).

IllegalParameterError - if the data chain is too long.

IllegalStateError - if the used channel is closed.

SecurityError - if the PIN to access the SS Applet was not verified.

IOError – if the entry could not be updated because of an incomplete write procedure.

(d) Method: byte[] read(String title)

This command reads and returns the byte stream of a data entry stored in the SE referenced by the title.

Parameters:

title - the title of the entry that shall be read. The maximum title length is 60. All characters must be supported by UTF-8.

Return value:

The data retrieved from the referenced entry. If the data does not exist in the SS entry referenced by the title then an empty byte array will be returned.

Errors:

IllegalParameterError - if the title is incorrect: bad encoding or wrong length (empty or too long).

IllegalStateError - if the used channel is closed.

SecurityError - if the PIN to access the SS Applet was not verified.

IOError – if the entry could not be read because of an incomplete read procedure.

(e) boolean exist(String title)

This command checks if the SS entry with the defined title exists.

Parameters:

title - the title of the entry that shall be checked. The maximum title length is 60. All characters must be supported by UTF-8.

Errors:

IllegalParameterError - if the title is incorrect: bad encoding or wrong length (empty or too long).

IllegalStateError - if the used channel is closed.

SecurityError - if the PIN to access the SS Applet was not verified.

Return value:

True if the entry with the defined title exists. False if the entry does not exist.

(f) Method: boolean delete(String title)

This command deletes the SS entry referenced by the title. If the entry does not exist, nothing will be done.

Parameters:

title - the title of the entry that shall be deleted. The maximum title length is 60. All characters must be supported by UTF-8.

Errors:

IllegalParameterError - if the title is incorrect: bad encoding or wrong length (empty or too long).

IllegalStateError - if the used channel is closed.

SecurityError - if the PIN to access the SS Applet was not verified.

Return value:

True if the entry with the defined title is deleted. False if the entry does not exist.

(g) Method: void deleteAll()

This command deletes all SS entry referenced. If no entries exist nothing will be done.

Errors:

IllegalStateError - if the used channel is closed.

SecurityError - if the PIN to access the SS Applet was not verified.

(h) Method: String[] list()
This command returns an entry list with all title-identifiers. The title is intended for the users to identify and to reference the SS entries.

Return value:
A list of titles of all entries located in SS. An empty list will be returned if no entries exist in the SS.

Errors:
IllegalStateError - if the used channel is closed.
SecurityError - if the PIN to access the SS Applet was not verified.

7.10.2   Secure Storage APDU Interface
The SS Applet has to provide this APDU command interface for adding entries to the SS and deleting entries from the SS. Each SS entry is a container for a SS data record consisting of a title and data attribute. The attribute title identifies the SS entry with a user readable text and must be unique. The SS entry title shall be defined by the user before the SS entry is created. Thus the user can identify the created SS entry within the SS afterwards. The data attribute contains the sensitive data which has to be stored into the SS. Besides the title, each SS entry can also be identified with a unique ID which is generated by the SS during the creation and has to be used to reference an SS entry within the SS. The title and ID must be unique within a SS, as each entry can be referenced by either the title or the ID.
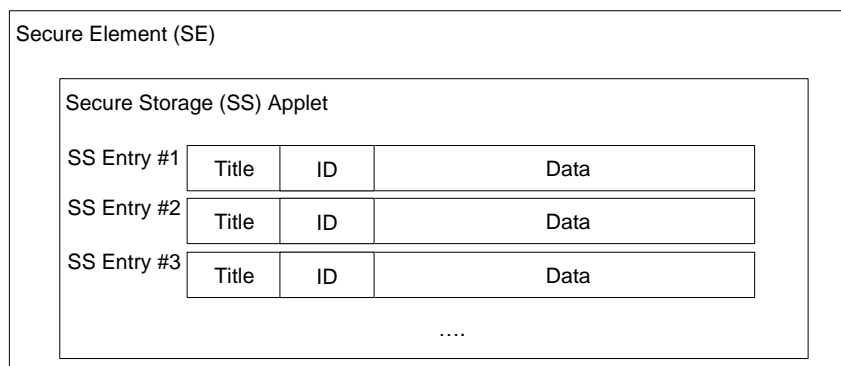
Figure 7-11: Secure Storage Applet overview

(a) CREATE SS ENTRY Command Message
The CREATE SS ENTRY command can be used to create an entry in the SS. Each entry requires a unique title which must be specified in the command.

The CREATE SS ENTRY command message shall be coded according to the following table:

Table 7-1: CREATE SS ENTRY Command Message

Code	Value	Meaning
CLA	'80'	
INS	'E0'	CREATE SS ENTRY
P1 P2	'00 00'	
LC	Length of title	
DATA	Title	Title of the new entry in UTF-8. This title must be unique within the SS. If the defined title does already exist in the SS then the error code '6A 80' will be returned.
LE	2	

(b) CREATE SS ENTRY Response Message

The CREATE SS ENTRY response shall contain a data field with response code '90 00' (successful operation) or an error response code.

Table 7-2: CREATE SS ENTRY Response Data

Value	Meaning	Presence
ID	The ID of the new entry created in the SS. This ID can be used to reference an SS entry. The length of this ID is always 2 bytes.	Mandatory

Table 7-3: CREATE SS ENTRY Response Code

SW1	SW2	Meaning
'6A'	'80'	Incorrect values in the command data (if the defined title does already exist)
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'67'	'00'	Wrong length in LC
'6A'	'86'	Incorrect P1 P2
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class
'65'	'81'	Memory failure (if the creation of the entry fails due to memory issues)
'6A'	'84'	Not enough memory space (if not enough memory resources are available)

(c) DELETE SS ENTRY Command Message

The DELETE SS ENTRY command can be used to delete an entry from the SS. The entry referenced in this command by an ID must exist in the SS otherwise an error code will be returned.

The DELETE SS ENTRY command message shall be coded according to the following table:

sim**alliance**

Table 7-4: DELETE SS ENTRY Command Message

Code	Value	Meaning
CLA	'80'	
INS	'E4'	DELETE SS ENTRY
P1 P2	P1: ID high byte P2: ID low byte	The ID of the SS entry which has to be deleted. If the SS entry couldn't be found '6A 88' is returned.
LC	-	
DATA	-	
LE	-	

(d) DELETE SS ENTRY Response Message
The DELETE SS ENTRY response shall contain a response code '90 00' (successful operation) or an error response code.

Table 7-5: DELETE SS ENTRY Response Code

SW1	SW2	Meaning
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6A'	'88'	Referenced data not found (if the referenced SS entry does not exist)
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class
'65'	'81'	Memory failure (if the operation fails due to memory issues)

(e) SELECT SS ENTRY Command Message
The SELECT SS ENTRY has to be used to select an SS ENTRY in the SS for a write or read operation.
The SELECT SS ENTRY command message shall be coded according to the following table:

Table 7-6: SELECT SS ENTRY Command Message

Code	Value	Meaning
CLA	'80'	
INS	'A5'	SELECT SS ENTRY
P1 P2	P1: Reference parameter P2: '00'	Reference parameter: Select ID('00'): Select the entry referenced by the ID Select First('01'): Select the first SS entry Select Next('02'): Select the next available SS entry
LC	2 or -	The length of the ID of the SS entry (only needed with P1= "Select ID")
DATA	ID or -	The ID of the SS entry which shall be selected (only needed with P1= "Select ID")
LE	'00'	

(f) SELECT SS ENTRY Response Message
The SELECT SS ENTRY response shall contain a data field with response code '90 00' (successful operation) or an error response code.

Table 7-7: SELECT SS ENTRY Response Data

Value	Meaning	Presence
Title	The title of the referenced SS entry in UTF-8.	Mandatory

Table 7-8: SELECT SS ENTRY Response Code

SW1	SW2	Meaning
'6A'	'80'	Incorrect values in the command data (if the data field has not a length of 2 bytes)
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6A'	'88'	Referenced data not found (if the referenced SS entry does not exist)
'67'	'00'	Wrong length in LC
'6A'	'86'	Incorrect P1 P2
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class

(g) PUT SS ENTRY DATA Command Message

The PUT SS ENTRY DATA command message can be used to store sensitive data into the currently selected SS entry. An SS entry can be selected with the command SELECT SS ENTRY DATA. For very long data the command PUT SS ENTRY DATA can be used iteratively by applying the command several times with an appropriate P1 parameter (first) and (next).

Note:
Before data can be stored into the SS entry with PUT SS ENTRY DATA, the data size has to be specified. Otherwise an error code will be returned.

The transmitted data will only be stored into the SS entry if all data parts are transferred (this means the sum of all transferred parts fits exactly to the defined data length). As long as the transferred data is not complete, the data has to be temporarily buffered within the SS application. If the succeeding APDU is not a PUT SS ENTRY DATA command or the succeeding PUT SS ENTRY DATA command does not contain the following data as expected, then the buffered data has to be discarded.

The PUT SS ENTRY DATA command message shall be coded according to the following table:

Table 7-9: PUT SS ENTRY DATA Command Message

Code	Value	Meaning
CLA	'80'	
INS	'DA'	PUT SS ENTRY DATA
P1 P2	P1: size (0), first (1), next(2) P2: '00'	size(0): The whole size of the data that shall be stored. first(1): DATA contains the first data part. next(2): DATA contains the next data part (append mode).
LC	Data length	
DATA	Data	P1=size(0): Defines the data size. P1=first(1) or next(2): Sensitive data (or a part of the data) which has to be stored to the currently selected SS entry.
LE	-	

sim alliance

(h) PUT SS ENTRY DATA Response Message

The PUT SS ENTRY ID response shall contain a response code '90 00' (successful operation) or an error response code.

Table 7-10: PUT SS ENTRY DATA Response Code

SW1	SW2	Meaning
'6A'	'80'	Incorrect values in the command data
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6A'	'88'	Referenced data not found (if no SS entry is currently selected)
'67'	'00'	Wrong length in LC
'6A'	'86'	Incorrect P1 P2 (if the defined P1/P2 are invalid or cannot be applied)
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class
'65'	'81'	Memory failure (if the defined data exceeds the defined size or a size was not defined)
'6A'	'84'	Not enough memory space (if not enough memory resources are available)

(i) GET SS ENTRY DATA Command Message

The GET SS ENTRY DATA command message can be used to retrieve data from the currently selected SS entry. An SS entry can be selected with the command SELECT SS ENTRY DATA. For very long data, the command GET SS ENTRY DATA can be used iteratively by applying the command several times with an appropriate P1 parameter (first) and (next). If the succeeding APDU is not a GET SS ENRTY DATA command, then an outstanding retrieve procedure must be reset by the SS application.

The GET SS ENTRY DATA command message shall be coded according to the following table:

Table 7-11: GET SS ENTRY DATA Command Message

Code	Value	Meaning
CLA	'80'	
INS	'CA'	GET SS ENTRY DATA
P1 P2	P1: size (0), first (1), next(2) P2: '00'	size(0): Response contains the whole size of the data that shall be read. first(1): Response contains the first data part. next(2): Response contains the next data part.
LC	-	
DATA	-	
LE	'00'	

(j) GET SS ENTRY DATA Response Message

The GET SS ENTRY DATA response shall contain a data field with response code '90 00' (successful operation) or an error response code.

Table 7-12: GET SS ENTRY DATA Response Data

Value	Meaning	Presence
Data	The data (or a part) of the currently selected SS entry. or Whole size of the data stored in the currently selected SS entry.	Mandatory

Table 7-13: GET SS ENTRY DATA Response Code

SW1	SW2	Meaning
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6A'	'86'	Incorrect P1 P2 (if the defined P1/P2 are invalid or cannot be applied)
'6A'	'88'	Referenced data not found (if no SS entry is currently selected)
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class
'65'	'81'	Memory failure (if further data is demanded but no further data exists).

(k) GET SS ENTRY ID Command Message

The GET SS ENTRY ID command message can be used to retrieve the ID of an SS entry referenced by the title.

The GET SS ENTRY ID command message shall be coded according to the following table:

Table 7-14: GET SS ENTRY ID Command Message

Code	Value	Meaning
CLA	'80'	
INS	'B2'	GET ENTRY ID
P1 P2	'00 00'	
LC	Length of title	
DATA	Title	Title of the entry
LE	'02'	

(l) GET SS ENTRY ID Response Message

The READ SS ENTRY ID response shall contain a data field with response code '90 00' (successful operation) or an error response code.

Table 7-15: GET SS ENTRY ID Response Data

Value	Meaning	Presence
ID	The ID of the entry in the SS referenced by the defined title. The length of this ID is always 2 bytes.	Mandatory

sim alliance

Table 7-16: GET SS ENTRY ID Response Code

SW1	SW2	Meaning
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6A'	'88'	Referenced data not found (if the referenced SS entry does not exist)
'6A'	'86'	Incorrect P1 P2
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class

(m) DELETE ALL SS ENTRIES Command Message

The DELETE ALL SS ENTRIES command can be used to delete all entries from the SS.

The DELETE ALL SS ENTRIES command message shall be coded according to the following table:

Table 7-17: DELETE ALL SS ENTRIES Command Message

Code	Value	Meaning
CLA	'80'	
INS	'E5'	DELETE ALL SS ENTRIES
P1 P2	'00 00'	
LC	-	
DATA	-	
LE	-	

(n) DELETE ALL SS ENTRIES Response Message

The DELETE SS ENTRIES response shall contain a response code '90 00' (successful operation) or an error response code.

Table 7-18: DELETE ALL SS ENTRIES Response Code

SW1	SW2	Meaning
'6A'	'82'	Security status not satisfied (if PIN verified state is not set)
'6D'	'00'	Invalid instruction
'6E'	'00'	Invalid class
'65'	'81'	Memory failure (if the operation fails due to memory issues)

7.10.3 Secure Storage APDU transfer

This chapter describes how the SS APDU interface has to be applied for performing the SS service operations provided by the SecureStorageProvider.

Note:

All SS operations have to be realised in an atomic way. This means if an error occurs during a SS operation (e.g. if an error occurs on a certain APDU) all modifications made on the SS (in the previous steps within a SS operation) have to be reversed.

simalliance

(a) Create operation

The create method includes the creation and selection of an SS entry with a succeeding SS entry data update. Following steps are needed:

- CREATE SS ENTRY (tile) creates the SS entry in the SS with the defined title.
- SELECT SS ENTRY (ID) selects the SS entry in the SS for the data update.
- PUT SS ENTRY DATA (size) to define the data size.
- PUT SS ENTRY DATA (data) writes the data to the selected SS entry. A long data chain (which cannot be transferred via one APDU command) can be written by applying the command iteratively by using PUT SS ENTRY DATA(P1P2=NEXT) several times after performing PUT SS ENTRY DATA(P1P2=FIRST).

```
void create(String title, byte[] data)
                              ID = CREATE SS ENTRY (title)
                              SELECT SS ENTRY (ID)

                              PUT SS ENTRY DATA (size)

                              PUT SS ENTRY DATA (FIRST, data first part)

                              PUT SS ENTRY DATA (NEXT, data next part)
```

Figure 7-12: Create SS entry operation

Note:

If an error occurs during the create operation, all previous steps must be reversed to obtain the same SS state as before. For example, if the creation of the SS entry was successful but the storage of the SS entry data fails, then this newly created SS entry has to be deleted again.

(b) Update operation

The update method includes the selection of an SS entry with a succeeding SS entry data update. The following steps are needed:

- GET ENTRY ID (title) returns the internal SS entry ID to the defined title.
- SELECT SS ENTRY (ID) selects the SS entry in the SS for the data update.
- PUT SS ENTRY DATA (size) to define the data size.
- PUT SS ENTRY DATA (data) writes the data to the selected SS entry. A long data chain (which cannot be transferred via one APDU command) can be written by applying the command iteratively by using PUT SS ENTRY DATA (P1P2=NEXT) several times after performing PUT SS ENTRY DATA (P1P2=FIRST).
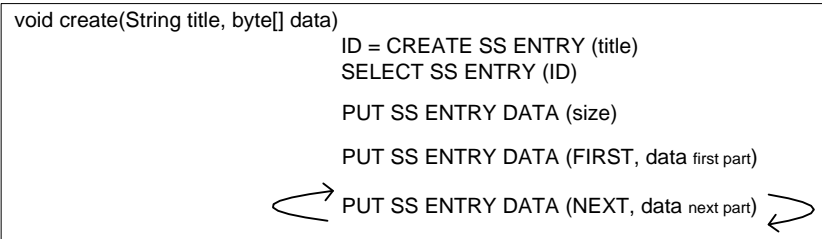
```
void update(String title, byte[] data)
                              ID = GET SS ENTRY ID (title)
                              SELECT SS ENTRY (ID)

                              PUT SS ENTRY DATA (SIZE, size)

                              PUT SS ENTRY DATA (FIRST, data first part)

                              PUT SS ENTRY DATA (NEXT, data next part)
```
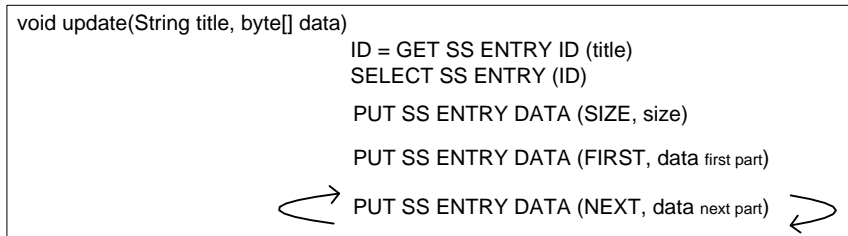
Figure 7-13: Update SS entry operation

Note:

If an error occurs during the update operation, all previous steps must be reversed to obtain the same SS state as before. For example, if the update of some data parts was

successful but the update of a following data part fails then the SS entry has to be set to the previous state (e.g. by reassigning the previously stored data to the SS entry).

(c) Read operation

The read method includes the selection of an SS entry with a succeeding read SS entry data operation. The following steps are needed:

- GET ENTRY ID (title) returns the internal SS entry ID to the defined title.
- SELECT SS ENTRY (ID) selects the SS entry in the SS for the read operation.
- GET SS ENTRY DATA (size) to determine the whole size of the data that shall be read.
- GET SS ENTRY DATA (data) reads the data to the selected SS entry. A long data chain (which cannot be transferred via one APDU command) can be read by applying the command iteratively by using GET SS ENTRY DATA (P1P2=NEXT) several times after performing GET SS ENTRY DATA (P1P2=FIRST).
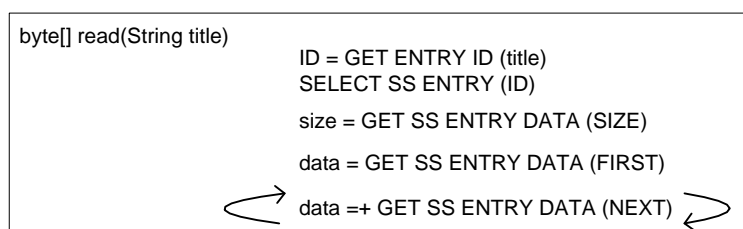
```
byte[] read(String title)
                        ID = GET ENTRY ID (title)
                        SELECT SS ENTRY (ID)
                        size = GET SS ENTRY DATA (SIZE)
                        data = GET SS ENTRY DATA (FIRST)
                        data =+ GET SS ENTRY DATA (NEXT)
```

Figure 7-14: Read SS entry operation

(d) List operation

The list method includes an iterative selection of all SS entries. The following steps are needed:

- SELECT SS ENTRY (FIRST) selects the first SS entry and returns its title.
- SELECT SS ENTRY (NEXT) selects the next SS entry and returns its title. This command has to be applied iteratively until all SS entry titles are retrieved.
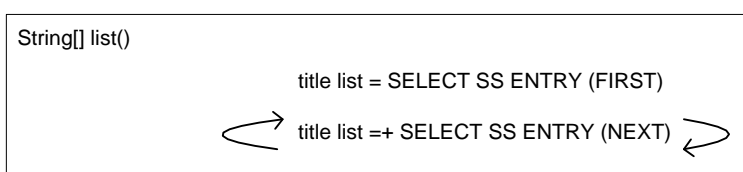
```
String[] list()
                        title list = SELECT SS ENTRY (FIRST)
                        title list =+ SELECT SS ENTRY (NEXT)
```

Figure 7-15: List SS entries operation

(e) Delete operation

The delete method includes a delete operation. The following steps are needed:

- GET ENTRY ID (title) returns the internal SS entry ID to the defined title.
- DELETE SS ENTRY (ID) deletes the SS entry referenced by the defined ID.

```
boolean delete(String title)
                        ID = GET ENTRY ID (title)
                        DELETE SS ENTRY (ID)
```

Figure 7-16: Delete SS entry operation

(f) Delete all operation

The delete all method includes a delete operation which deletes all entries from the SS. The following steps are needed:

- DELETE ALL SS ENTRIES

```
void deleteAll()
                        DELETE ALL SS ENTRIES
```

Figure 7-17: Delete all SS entries operation

(g) Exist operation

The exist method checks if a certain SS entry exists. The following steps are needed:

- GET ENTRY ID (title) returns the internal SS entry ID to the defined title.
- SELECT SS ENTRY (ID) indicates if the SS entry to the defined ID exists or not.

```
boolean exist(String title)
                    ID = GET ENTRY ID (title)
                    SELECT SS ENTRY (ID)
```

Figure 7-18: Exist SS entry operation

7.10.4   Secure Storage PIN protection

Before a SS command can be performed, a PIN verification has to be performed towards the SS Applet. The SS Applet allows the execution of an APDU command only after a successful PIN verification. Therefore the SS Applet stores an internal PIN verified state for each logical channel. Thus an established logical channel has to be authorised with a PIN verification before this channel can be used for the SecureStorageService. The internal PIN verified state for a channel is kept up until this channel is closed. If a SS APDU command is used without being authorised with a PIN, the SS Applet has to return an error code indicating the missing privilege.

To provide the SS access restriction based on PIN authentication, the SS Applet must provide the ISO/IEC 7816-4 commands VERIFY, CHANGE REFERENCE DATA and RESET RETRY COUNTER for supporting the Authentication Service methods verifyPin(byte[] pin), changePin (byte[] oldPin, byte[] new Pin) and resetPin(byte[] resetPin, byte[] newPin). The SS can also provide other PIN related ISO/IEC 7816-4 commands like DISABLE VERIFICATION REQUIREMENT, ENABLE VERIFICATION REQUIREMENT for allowing extended PIN management with the Authentication Service methods deactivatePin(byte[] pin) and activatePin(byte[] pin).

8. Recommendation for a Minimum Set of Functionality

SE access is essential for secure applications. As a result, any mobile device compliant with the Open Mobile API must provide access to all SEs on the device. The Transport API (with access to all SEs available in the device, e.g. SIM, microSD and eSE) is therefore mandatory for Open Mobile API compliant devices. A mobile device which has a SE with no access to the Transport API would not be considered compliant with the Open Mobile API.

The most common SEs today are SIM cards, secure SD cards or embedded SEs. But new SEs may emerge in the future. The SE provider interface is therefore mandatory, to ensure that the device can support new SEs in the future.

The Transport API shall support the maximum number of extended logical channels according to ISO 7816-4 specification (19 logical channels in addition to the basic channel).
In case the ATR is not available, or the ATR is available and indicates the support of 8 or more channels (including the basic channel), the API shall try to open logical channels, provided no error is indicated.
In case the ATR is available and indicates support of 7 or less channels (including the basic channel), the API shall manage as many logical channels as indicated by the ATR, either by using the ATR information or by opening logical channels, provided no error is indicated.

The Transport API shall support extended length APDU commands independent of the coding within the ATR.

These following components can be provided by device manufacturers or third parties:

- Discovery API
- Crypto API
- Secure Storage
- File Management
- Authentication
- PKCS#15

The mobile device should allow the installation of these services as an add-on API.

9. Secure Element Provider Interface

This provides a way to add and remove drivers for SEs at runtime, by installing or removing downloadable application packages. It enables communication with the SE.
The concrete API of such a SE Provider Interface is up to the individual implementer of the Open Mobile API and not defined in this document.

The following requirements, however, must be fulfilled:

- The implementation has to enforce that it is only used by the Transport Layer and not directly by mobile applications (so channel management and security mechanism in the transport layer cannot be bypassed).
- A reference implementation needs to be available.
- Existing SE Providers cannot be replaced by dynamically loaded providers.

10. Access Control

Access Control is used by the Transport and Service Layers. It is based on the signature of the mobile application and is enforced when accessing the Transport Layer.
The permission is based on access policies stored in the SE. These access policies define precisely which mobile application is allowed to access an applet installed in the SE.

The APIs in this specification will indicate errors (e.g. declaring security errors) when they are subject to the Access Control.
Access Control itself is defined by GlobalPlatform in the SE Access Control Working Group (see [9]).

11. History

Table 11-1: History

Version	Date	Author	Comment
1.0	28.02.2011	SIMalliance	Initial Release 1.0
1.01	16.03.2011	SIMalliance	Minor corrections
1.1	04.05.2011	SIMalliance	Clarifications for several functions in the transport layer
1.2	12.07.2011	SIMalliance	Correction for open Basic Channel
2.0	30.09.2011	SIMalliance	Adding descriptions of the service layer, corrections in the transport layer, adding getSelectResponse() to channel class
2.01	14.10.2011	SIMalliance	Minor corrections
2.02	4.11.2011	SIMalliance	Corrections in diagrams of the transport layer
2.03	19.06.2012	SIMalliance	Clarification on Chapter 10 (since GlobalPlatform SEAC spec is released), on 6.4.4. and 6.7.6 / 6.7.7
2.04	15.07.2013	SIMalliance	Clarification on Chapter 5, 6.2, 6.7.6, 6.7.7, 6.8.6, 7.1, 7.6.1, 7.6.3, 7.6.4, 7.6.5, 7.8.1 and chapter 8. Added 6.4.5, 6.8.7
2.05	28.01.2014	SIMalliance	Chapter 3, clarification on the namespace; Chapter 6.4.2: IllegalStateError added; Chapter 6.6.1 changed definition of reader name according request from GSMA; Chapter 6.8.6: clarification on handling of status words
3.0 draft4	26.08.2014	SIMalliance	Procedural interface added to chapter 6; reference header for Ansi-C defined in Annex; P2 parameter added for openBasicChannel and openLogicalChannel (previous interface without P2 is kept but deprecated)

sim alliance

Annex A:　Ansi-C Reference Header for Transport Procedural Interface

```c
/* omapi.h
 * Copyright (c) 2014 SIMalliance.org*/
#ifndef __omapi_h__
#define __omapi_h__

#ifdef __cplusplus
extern "C" {
#endif

/* platform specific mapping of SIMalliance data types */
#ifndef OMAPI_API
#define                         OMAPI_API
#endif
typedef int                 OMAPI_RESULT;
typedef int                 OMAPI_HANDLE;
typedef int                 Int;
typedef char *              String;
typedef unsigned char       Byte;
typedef enum { false, true }  Boolean;

/* SIMalliance return codes */
#define OMAPI_SUCCESS                       ((Int)0x00000000) /* No error was encountered */
#define OMAPI_GENERAL_ERROR                 ((Int)0x10000000) /* A general error occurred */
#define OMAPI_IO_ERROR                      ((Int)0x10000001) /* Communication error */
#define OMAPI_NO_SUCH_ELEMENT_ERROR         ((Int)0x10000002) /* No such element error */
#define OMAPI_ILLEGAL_STATE_ERROR           ((Int)0x10000003) /* Illegal state of execution error */
#define OMAPI_ILLEGAL_PARAMETER_ERROR       ((Int)0x10000004) /* Illegal or invalid parameter */
#define OMAPI_ILLEGAL_REFERENCE_ERROR       ((Int)0x10000005) /* Illegal reference */
#define OMAPI_OPERATION_NOT_SUPPORTED_ERROR ((Int)0x10000006) /* Operation not supported from SE */
#define OMAPI_SECURITY_ERROR                ((Int)0x10000007) /* Security Error blocks execution */
#define OMAPI_CHANNEL_NOT_AVAILABLE_ERROR   ((Int)0x10000008) /* No channel available */
#define OMAPI_NULL_POINTER_ERROR            ((Int)0x10000009) /* Null pointer not allowed */
```

simalliance

```
/* SIMalliance Open Mobile API */
OMAPI_API OMAPI_RESULT omapi_get_readers(OMAPI_HANDLE *phReaders, Int *pLength);
OMAPI_API OMAPI_RESULT omapi_get_version(String pVersion, Int *pLength);

OMAPI_API OMAPI_RESULT omapi_reader_get_name(OMAPI_HANDLE hReader, String pReader, Int *pLength);
OMAPI_API OMAPI_RESULT omapi_reader_is_secure_element_present(OMAPI_HANDLE hReader, Boolean *pIsPresent);
OMAPI_API OMAPI_RESULT omapi_reader_open_session(OMAPI_HANDLE hReader, OMAPI_HANDLE *phSession);
OMAPI_API OMAPI_RESULT omapi_reader_close_sessions(OMAPI_HANDLE hReader);

OMAPI_API OMAPI_RESULT omapi_session_get_reader(OMAPI_HANDLE hSession, OMAPI_HANDLE *phReader);
OMAPI_API OMAPI_RESULT omapi_session_get_atr(OMAPI_HANDLE hSession, Byte *pAtr, Int *pLength);
OMAPI_API OMAPI_RESULT omapi_session_close(OMAPI_HANDLE hSession);
OMAPI_API OMAPI_RESULT omapi_session_is_closed(OMAPI_HANDLE hSession, Boolean *pIsClosed);
OMAPI_API OMAPI_RESULT omapi_session_close_channels(OMAPI_HANDLE hSession);
OMAPI_API OMAPI_RESULT omapi_session_open_basic_channel(OMAPI_HANDLE hSession, Byte *AID, Int length, Byte
P2, OMAPI_HANDLE *phChannel);
OMAPI_API OMAPI_RESULT omapi_session_open_logical_channel(OMAPI_HANDLE hSession, Byte *AID, Int length,
Byte P2, OMAPI_HANDLE *phChannel);

OMAPI_API OMAPI_RESULT omapi_channel_close(OMAPI_HANDLE hChannel);
OMAPI_API OMAPI_RESULT omapi_channel_is_basic_channel(OMAPI_HANDLE hChannel, Boolean *pIsBasicChannel);
OMAPI_API OMAPI_RESULT omapi_channel_is_closed(OMAPI_HANDLE hChannel, Boolean *pIsClosed);
OMAPI_API OMAPI_RESULT omapi_channel_get_select_response(OMAPI_HANDLE hChannel, Byte *pSelectResponse, Int
*pLength);
OMAPI_API OMAPI_RESULT omapi_channel_get_session(OMAPI_HANDLE hChannel, OMAPI_HANDLE *phSession);
OMAPI_API OMAPI_RESULT omapi_channel_transmit(OMAPI_HANDLE hChannel, Byte *pCommand, Int cmdLength, Byte
*pResponse, Int *pRspLength);
OMAPI_API OMAPI_RESULT omapi_channel_transmit_receive_response(OMAPI_HANDLE hChannel, Byte *pResponse, Int
*pRspLength);
OMAPI_API OMAPI_RESULT omapi_channel_select_next(OMAPI_HANDLE hChannel, Boolean *pSuccess);

#ifdef __cplusplus
}
#endif
```

Security, Identity, Mobility

```
#endif
```