GlobalPlatform Technology

TPS Client API Specification

Version 1.0

Public Release

March 2025

Document Reference: GPP_SPE_009

Contents

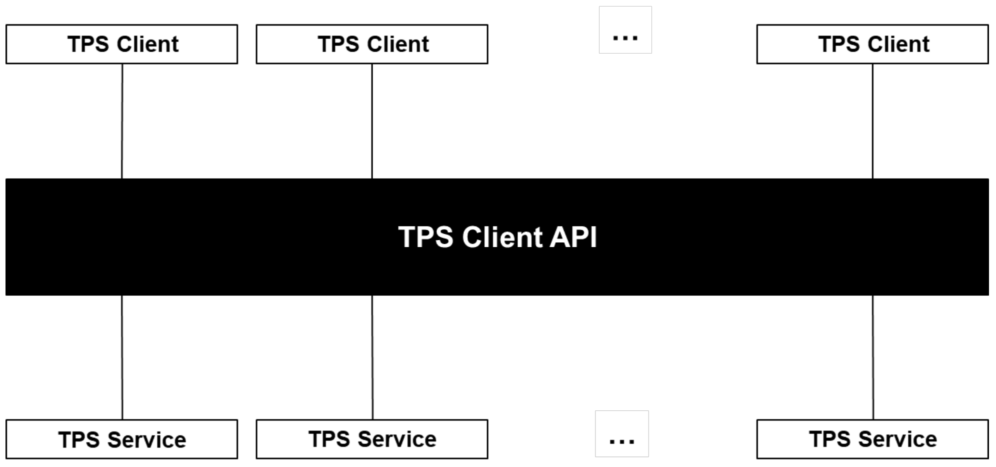
Tables

Figures

1 INTRODUCTION

This specification defines the TPS Client API, a communications API for connecting *TPS Clients* with *TPS Services* where the TPS Client connecting to a TPS Service can be either an *Application* or another TPS Service. The TPS Client API provides a C language interface used to discover, open a session, communicate, and close the session with a TPS Service. The details of TPS Services and the communication protocols to communicate with them are specified in separate documents.

Figure 1-1: Active Modules of TPS Client API



If you are implementing this specification and you think it is not clear on something:

1. **Check with a colleague.**

And if that fails:

2. **Contact GlobalPlatform at TPS-Client-API-issues-GPP_SPE_009@globalplatform.org**

1.1 Audience

This document is suitable for software developers implementing:

- Applications that use TPS Services
- TPS Services
- the TPS Client API and the communications infrastructure required to access TPS Services

As this API is the base layer upon which higher level protocols providing TPS Services are built, it will also be of interest to developers of future TPS Service specifications that build higher-level APIs on top of it.

1.2 IPR Disclaimer

Attention is drawn to the possibility that some of the elements of this GlobalPlatform specification or other work product may be the subject of intellectual property rights (IPR) held by GlobalPlatform members or others. For additional information regarding any such IPR that have been brought to the attention of GlobalPlatform, please visit https://globalplatform.org/specifications/ip-disclaimers/. GlobalPlatform shall not be held responsible for identifying any or all such IPR, and takes no position concerning the possible existence or the evidence, validity, or scope of any such IPR.

1.3 References

This section lists references applicable to this specification. The latest version of each reference applies unless a publication date or version is explicitly stated.

Table 1-1: Normative References

Standard / Specification	Description	Ref
GPD_SPE_010	GlobalPlatform Technology TEE Internal Core API Specification	[TEE Core]
IETF RFC 2119	Key words for use in RFCs to Indicate Requirement Levels	[RFC 2119]
IETF RFC 8174	Amendment to RFC 2119	[RFC 8174]
ISO/IEC 9899:1999	Programming languages – C	[C99]
Semantic Versioning	Semantic Versioning (https://semver.org/)	[Sem Ver]

Table 1-2: Informative References

Standard / Specification	Description	Ref
GPC_SPE_034	GlobalPlatform Technology Card Specification	[GPCS]
GPD_SPE_007	GlobalPlatform Technology TEE Client API Specification	[TEE Client]
GPD_SPE_075	GlobalPlatform Technology Open Mobile API Specification	[OMAPI]
IETF RFC 4122	A Universally Unique IDentifier (UUID) URN Namespace	[RFC 4122]
TCG FAPI	Trusted Computing Group Feature API	[FAPI]

1.4 Terminology and Definitions

The following meanings apply to SHALL, SHALL NOT, MUST, MUST NOT, SHOULD, SHOULD NOT, and MAY in this document (refer to [RFC 2119] as amended by [RFC 8174]):

- **SHALL** indicates an absolute requirement, as does **MUST**.
- **SHALL NOT** indicates an absolute prohibition, as does **MUST NOT**.
- **SHOULD** and **SHOULD NOT** indicate recommendations.
- **MAY** indicates an option.

Note that as clarified in the [RFC 8174] amendment, lower case use of these words is not normative.

Selected terms used in this document are included in Table 1-3.

Table 1-3: Terminology and Definitions

Term	Definition
Applet	General term for a Secure Element application: An application as described in GlobalPlatform Card Specification ([GPCS]) that is installed in the SE and runs within the SE.
Application	Device/terminal/mobile application. An application that is installed in and runs within the Regular Execution Environment.
Binding	A mapping between a Language Specific API and the TPS Client API that translates Language Specific API calls to TPS Service Protocol messages specified in a TPS Service specification, and vice versa.
Communication stack	The mechanisms by which a TPS Service present in a Secure Component is accessed via the TPS Client API. For more information, see section 3.2.7.
Connector	See *TPS Client Connector*.
Device	An end-user product that includes at least one Platform.
Execution Environment	An environment that hosts and executes software. This could be a REE, with hardware hosting Android, Linux, Windows, an RTOS, or other software; it could be a Secure Element or a TEE.
Implementation	The TPS Client API implementation and underlying Communication stack implementations enabling the usage of TPS Services supported by various Secure Components.
Language Specific API	An API that enables the usage of a TPS Service using a native programmatic interface for a specific programming language. See also *Binding*.
Platform	One computing engine and executable code that provides a set of functionalities. SE, TEE, and REE are examples of platforms. In the context of this document, Platform is used specifically to denote the Platform on which the TPS Client API executes, rather than any other Platform on the Device.

Term	Definition
Regular Execution Environment (REE)	An Execution Environment comprising at least one Regular OS and all other components of the device (IC packages, other discrete components, firmware, and software) that execute, host, and support the Regular OSes (excluding any Secure Components included in the device). From the viewpoint of a Secure Component, everything in the REE is considered untrusted, though from the Regular OS point of view there may be internal trust structures. (Formerly referred to as a *Rich Execution Environment (REE)*.) Contrast *Trusted Execution Environment (TEE).*
Regular OS	An OS executing in a Regular Execution Environment. May be anything from a large OS such as Linux down to a minimal set of statically linked libraries providing services such as a TCP/IP stack. (Formerly referred to as a *Rich OS* or *Device OS*.)
Secure Component	A trusted and secure component that is able to provide enhanced security compared to the Regular Operating System. Examples include GlobalPlatform's Secure Element and Trusted Execution Environment, and Trusted Computing Group's Trusted Platform Module.
Secure Element (SE)	A tamper-resistant secure hardware component that is used in a device to provide the security, confidentiality, and multiple application environment required to support various business models. May exist in any form factor, such as embedded or integrated SE, SIM/UICC, smart card, smart microSD, etc.
TPS Client	An entity that uses the TPS Client API to discover and communicate with a TPS Service. A TPS Client can be either an Application or another TPS Service.
TPS Client API	The API defined in this specification, which enables generic mechanisms for discovering and communicating with a TPS Service.
TPS Client Connector	An interface to a Communication stack for a particular type of Secure Component.
TPS Operation	An operation that is executed by a TPS Service upon a request from a TPS Client. A TPS Operation consists of one or more TPS Transactions.
TPS Service	A service in a Secure Component, providing a service to entities in the operating system; accessed using a TPS Service Protocol that is specified in a TPS Service specification.
TPS Service Name	Uniquely identifies a TPS Service implementation.
TPS Service Protocol	A protocol that is used to communicate with the TPS Service; consists of a set of TPS Operations.
TPS Service request message	A protocol message specified by a TPS Service specification. It is constructed and sent by the TPS Client to the TPS Service using the TPS Client API.

Term	Definition
TPS Service response message	A protocol message specified by a TPS Service specification. It is constructed and sent by the TPS Service to the TPS Client in response to a TPS Service request message.
TPS Session	An abstraction of a logical connection between a TPS Client and a TPS Service instance.
TPS Transaction	A single exchange of messages between the TPS Client and TPS Service: a TPS Service request message created and sent by a TPS Client to a TPS Service, and a TPS Service response message created by the TPS Service and sent to the TPS Client in response to the TPS Service request message.
Trusted Execution Environment (TEE)	An Execution Environment that runs alongside but isolated from Execution Environments outside of the TEE. A TEE has security capabilities and meets certain security-related requirements: It protects TEE assets against a set of defined threats which include general software attacks as well as some hardware attacks, and defines rigid safeguards as to data and functions that a program can access. There are multiple technologies that can be used to implement a TEE, and the level of security achieved varies accordingly. Contrast *Regular Execution Environment (REE).*
Trusted Platform Module (TPM)	A computer chip (microcontroller) that can securely store artifacts used to authenticate the platform. These artifacts can include passwords, certificates, or encryption keys. A TPM can also be used to store platform measurements that help ensure that the platform remains trustworthy.
UUIDv5	In this document, UUIDv5 is used to denote a name-based Universally Unique Identifier constructed using SHA-1 hashing, as described in [RFC 4122].

1.5 Abbreviations

Table 1-4: Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
CBOR	Concise Binary Object Representation
FFI	Foreign Function Interface
OS	Operating System
REE	Regular Execution Environment
RFU	Reserved for Future Use
SE	Secure Element
TEE	Trusted Execution Environment
TPS	Trusted Platform Service

1.6 Revision History

GlobalPlatform technical documents numbered *n*.0 are major releases. Those numbered *n*.1, *n*.2, etc., are minor releases where changes typically introduce supplementary items that do not impact backward compatibility or interoperability of the specifications. Those numbered *n.n*.1, *n.n*.2, etc., are maintenance releases that incorporate errata and clarifications; all non-trivial changes are indicated, often with revision marks.

Table 1-5: Revision History

Date	Version	Description
March 2025	v1.0	Public Release

2 OVERVIEW

This specification defines a communications API for connecting TPS Clients with TPS Services where the TPS Client connecting to a TPS Service can be either an Application or another TPS Service. The TPS Client API provides a C language interface and an optional Rust language interface that can be used to discover, open a session, communicate, and close the session with a TPS Service.

The TPS Client API executes on a Platform. It has been designed to be implementable on many possible systems. In particular, the TPS Client API is designed to be implemented both on many instances of REE and on a GlobalPlatform TEE.

The details of TPS Services and the communication protocols to communicate with the TPS Services are specified in separate specifications.

2.1 Standardization Scope

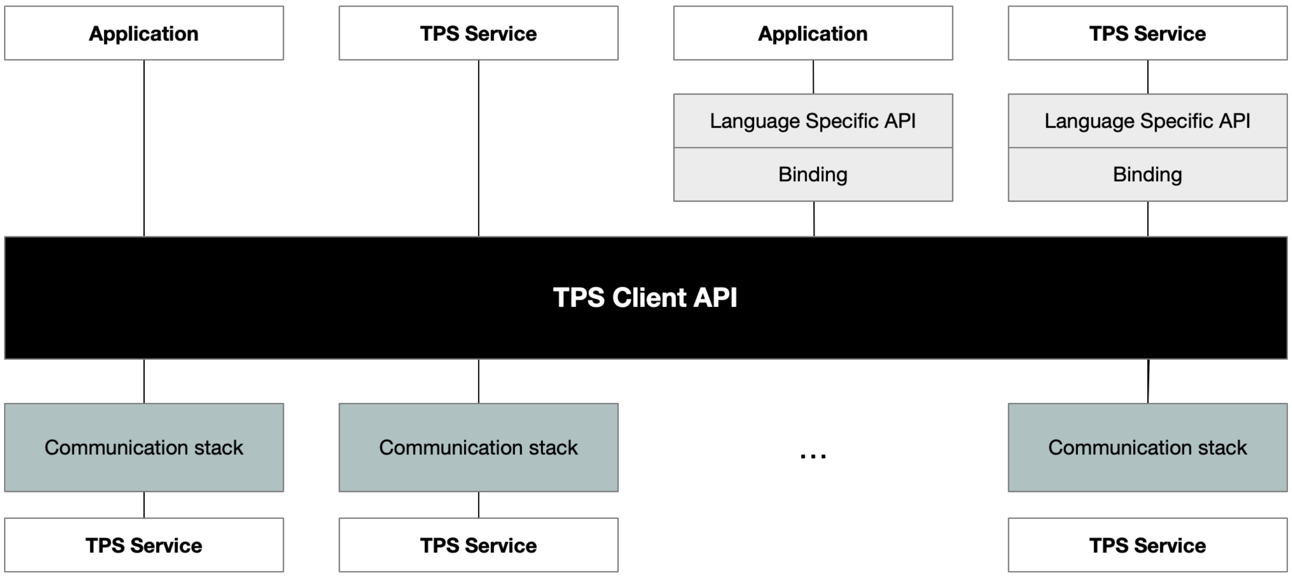
Instead of trying to standardize a single monolithic API that covers a significant proportion of the interactions between TPS Entities and TPS Services, the approach of the GlobalPlatform standardization effort is modular. The TPS Client API covered by this specification concentrates on the interface to enable efficient communications between a TPS Client (i.e. an Application or a TPS Service) and a TPS Service.

Higher level specifications and protocol layers providing TPS Services can be built on top of the foundation provided by the TPS Client API. These interfaces are out of scope of this specification.

2.2 TPS Client API Architecture

The relationships between the system components related to the TPS Client API are outlined in the block architecture in Figure 2-1. The TPS Client API connects a TPS Client with a TPS Service. A TPS Client can be either an Application or another TPS Service. TPS Services may be used via a Language Specific API implemented using a Binding between the Language Specific API and the TPS Service. The Binding uses the TPS Client API to make use of services provided by the TPS Service, which are then provided to the TPS Client (an Application or another TPS Service) via the Language Specific API.

Figure 2-1: TPS Client API Architecture



The TPS Client API is connected to one or more TPS Services, each available to the TPS Client API via a *Communication stack*. The Communication stack is used to establish the communication channel between the TPS Client API and the TPS Service implementation.

The TPS Client API is the main component of this architecture. It is used to establish a *TPS Session* between a TPS Client and a TPS Service and subsequently to execute *TPS Operations* through the session. The session can be viewed as a connection, or as a channel between the client and the service through which a set of operations can be executed. A TPS Operation consists of one or more *TPS Transactions*, which are request-response pair messages instructing a TPS Service to do operations specific to the service. (TPS Session, TPS Operation, and TPS Transaction are further discussed in section 3.2.)

3 PRINCIPLES AND CONCEPTS

This section explains the underlying principles and concepts of the TPS Client API in detail and describes how each class of features should be used.

3.1 Design Principles

Note: An optional, equivalent native Rust language external interface is provided in Annex A.

The key design principles of the TPS Client API are:

- **C language API**

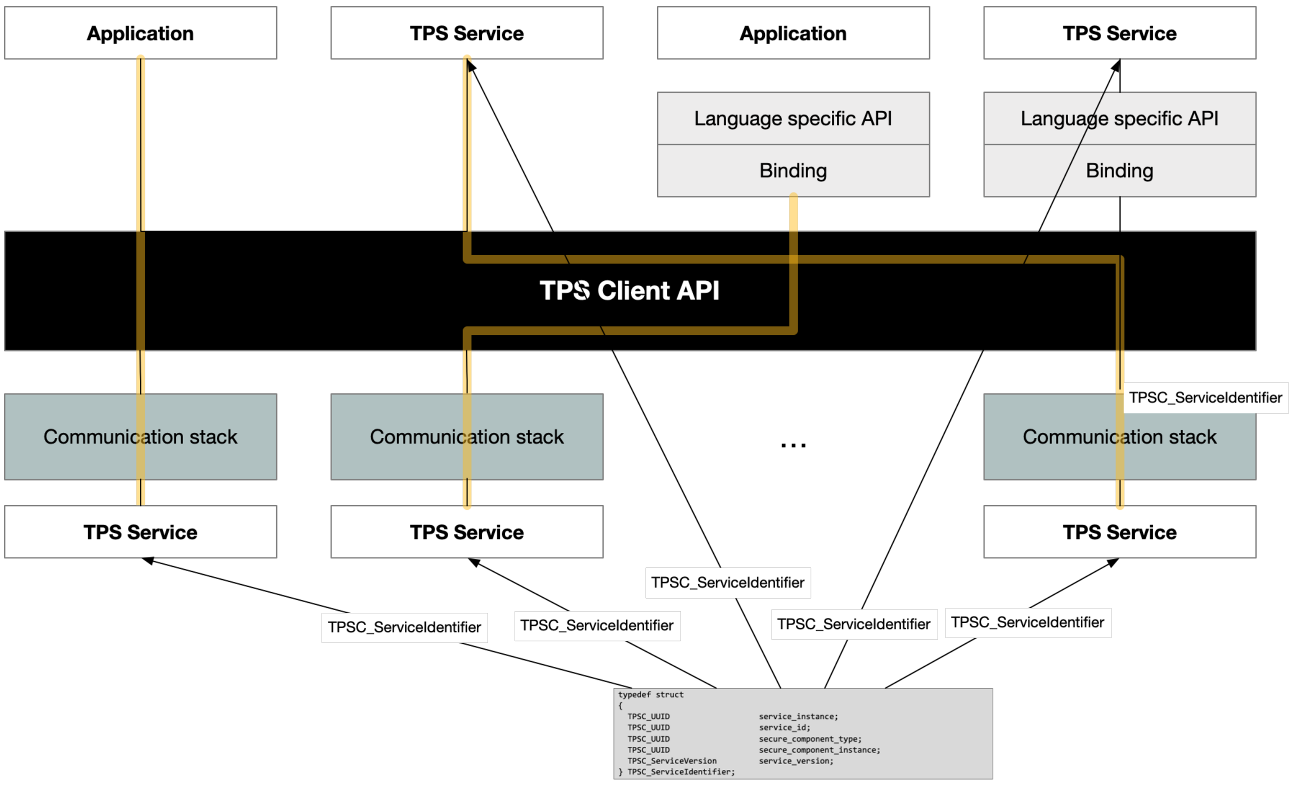
  Note: While a C language API is presented to clients, this does not constrain the programming environment used for a given implementation except that it must be able to expose the C language API described in this document.

  - C is the common denominator for the application frameworks and operating systems hosting Applications that use the TPS Client API and can be supported by almost all other platform programming language options.
- **Blocking functions**
  - Most Application developers are familiar with synchronous functions that block while waiting for the underlying task to complete before returning to the calling code. An asynchronous interface is hard to design, hard to port to Regular OS environments, and is generally difficult for developers familiar with synchronous APIs to use.
  - A mechanism to support cancellation of blocking API functions is optional. Where the OS supports multi-threading, implementations SHOULD support cancellation.
- **Source-level portability**
  - To enable compile-time and design-time optimization, this specification places no requirement on binary compatibility beyond that provided by the OS. Application developers may need to recompile their code against an *implementation-provided* version of the TPS Client API headers and libraries to build correctly on that implementation.
- **Specify both the communication mechanism and the format of messages**
  - This API focuses on defining the underlying communications channel. TPS Service specifications will define the format of the messages that are passed over the channel.

3.2 Fundamental Concepts

This section outlines the behavior of the TPS Client API and introduces key concepts and terminology. Figure 3-1 shows these graphically.

Figure 3-1: TPS Entities and Concepts



```
typedef struct
{
  TPSC_UUID               service_instance;
  TPSC_UUID               service_id;
  TPSC_UUID               secure_component_type;
  TPSC_UUID               secure_component_instance;
  TPSC_ServiceVersion     service_version;
} TPSC_ServiceIdentifier;
```

3.2.1 TPS Client

A TPS Client is an entity that uses the TPS Client API to access services provided by a TPS Service. A TPS Client can be an Application or another TPS Service.

3.2.2 TPS Service

A TPS Service is an entity that provides a service to TPS Clients. A TPS Service is discovered, connected to, communicated with, and disconnected from using the TPS Client API.

3.2.3 TPS Service Identifiers

The TPS Service Identifier allow a client to select which TPS Services provided by a Platform it might wish to use. It enables use cases such as:

- A client application wishes to use the same instance of a TPS Service whenever it runs.
- A client application wishes to select one from any of the available instances of a particular service.
- A client application wishes to select a TPS Service residing on a particular type of Secure Component.
- A client application wishes to select a TPS Service residing on a specific instance of a specific type of Secure Component.
- A client application wishes to select a TPS Service with at least a specified version.

3.2.3.1 Elements of the TPS Service Identifier

The TPS Service Identifier is composed of the following information:

- An identifier, `tps-service-id`, for the functionality provided by the TPS Service
- An identifier, `tps-service-version`, for the version of the TPS Service
- An identifier, `tps-secure-component-type`, indicating the type of security environment supporting the TPS Service
- A Platform unique identifier, `tps-secure-component-instance`, for the security environment instance. This can be used, for example, to differentiate between multiple TEEs on a Platform.
- A Platform unique identifier, `tps-service-instance`, for a specific instance of a TPS Service on a particular security environment on the Platform

3.2.3.2 UUIDs

Many of the values that define a TPS Service are presented as UUID [RFC 4122] values.

Note: For convenience of representation, informative examples in this document use the string representation defined in [RFC 4122] with the urn prefix removed.

Implementers are advised that the TPS APIs represent UUIDs as an array of 16 bytes of type `TPSC_UUID`. See section 4.3.11.

Except where stated otherwise, UUID types in this document are constructed using the Algorithm for Creating a Name-based UUID using SHA-1 hashing, described in [RFC 4122] and often abbreviated to UUIDv5. This constructs values from a UUID Namespace and a Name.

3.2.3.2.1 UUID Namespace

For TPS Services, where a UUIDv5 is required, the UUIDv5 `namespace` SHALL be set to:

```
9913673c-233e-422c-8213-1ec1f74936e8
```

This value is a randomly generated UUIDv4 and serves to ensure a low probability of collision between UUIDs describing TPS Services and other UUIDv5 namespaces.

3.2.3.2.2 Defining the tps-service-name in a UUID

This specification generally uses UTF-8 strings to define `tps-service-name` values to be used as UUID names. Where the UUID name is implementation-defined, the name can be constructed from any type that has a canonical transformation into an array of bytes.

To reduce the probability of UUID value collisions, there are rules constraining UUID names defined as strings and UUID names defined otherwise.

Names defined as UTF-8 Strings

One of the prefixes below SHALL be prepended to all UUID names defined as UTF-8 strings.

- One of the prefixes "GPP", "GPD", “GPT” and "GPC" MUST be selected for TPS Services defined by GlobalPlatform specifications. These prefixes are reserved and MUST NOT be used by bodies other than GlobalPlatform to define a name within the context of a TPS Service.
- The prefix "TCG" MUST be used for TPS Services defined by Trusted Computing Group specifications. This prefix is reserved and MUST NOT be used by bodies other than the Trusted Computing Group to define a name within the context of a TPS Service.
- The prefix "STD" SHOULD be used for TPS Services defined in specifications published by other standards bodies and industry groups. Bodies SHOULD take reasonable care to avoid name collisions, for example by including the name of the standards body in the name.
- The prefix "VND" is reserved for proprietary TPS Service definitions. Proprietary definitions SHOULD include the name of the defining entity to reduce the chance of naming collisions.

Names not defined as UTF-8 Strings

- There MUST be a canonical method to transform the type used as the base for the name into a sequence of bytes.
- The sequence of bytes generated from the type MUST NOT start with any of the following reserved sequences of bytes (these correspond to the reserved UTF-8 string prefixes):
  - `[0x47, 0x50, 0x50]`
  - `[0x47, 0x50, 0x44]`
  - `[0x47, 0x50, 0x54]`
  - `[0x47, 0x50, 0x43]`
  - `[0x54, 0x43, 0x47]`
  - `[0x53, 0x54, 0x44]`
  - `[0x56, 0x4e, 0x44]`

3.2.3.3 tps-service-id

The `tps-service-id` allows a client to determine the class of functionality provided by a TPS Service instance.

Any specification defining a TPS Service SHALL define `tps-service-name` to uniquely identify the service within the set of all TPS Services.

The `tps-service-id` is a UUIDv5 as defined in section 3.2.3.2 where the `name` field is set to `tps-service-name`.

3.2.3.3.1 Informative Examples

The informative examples below are intended to assist specification writers in defining an interoperable `tps-service-name`.

```
tps-service-name = "GPP ROT13"
```

`tps-service-id` is the generated UUIDv5: `87bae713-b08f-5e28-b9ee-4aa6e202440e`

```
tps-service-name = "VND Acme Detonator Service"
```

`tps-service-id` is the generated UUIDv5: `bd04103d-9ff4-5b40-a8f9-fdffc07ffce8`

```
tps-service-name = "STD StandardsBody ServiceName"
```

`tps-service-id` is the generated UUIDv5: `4876bf7f-367a-5e30-bd7e-a0d8bd66b77b`

3.2.3.4 tps-service-version

`tps-service-version` represents the version of the service, following Semantic Versioning ([Sem Ver]) conventions. It has three parts: the Major Version; Minor Version, and Patch Version.

Where `tps-service-version` is expressed as a string, e.g. in the derivation of new UUIDs, it shall be serialized as a sequence of concatenated 32bit hexadecimal values including leading zeroes.

As an example, `tps-service-version` where Major Version is 2, Minor Version is 13, and Patch version is 21 is expressed as the string `"000000020000000d00000015"`.

3.2.3.4.1 Major Version

The Major Version MUST be incremented if any backward-incompatible change is made to the service API.

If GlobalPlatform manages the TPS Service specification, the Major Version of the service MUST match the major version of the specification. That is, any backward incompatible change to a TPS Service requires an increment to the major version of the specification.

An exception to the above rules is made for Major Version 0. This version is used for initial development of a service API and indicates that it is unstable. This means that anything MAY change at any time.

3.2.3.4.2 Minor Version

The Minor Version SHOULD be incremented if any backward-compatible change is made to the service API.

If GlobalPlatform manages the TPS Service specification, the Minor Version of the service MUST match the minor version of the specification. That is, any backward compatible change to a TPS Service requires an increment to the minor version of the specification.

3.2.3.4.3 Patch Version

The Patch Version SHOULD be incremented if any backward-compatible change is made to the service API.

The Patch Version is used to distinguish between different versions of work in progress, such as a draft proposal. Patch Version additions and changes are unstable and may change at any time. End-users of the API SHOULD NOT rely on the behavior of Patch Versions.

3.2.3.4.4 Service Version Constraints

During service discovery, the caller may wish to limit acceptable service versions. The TPS Client API provides a mechanism to enable this in which the client specifies:

- The lowest acceptable version of the service.
  - For inclusive bounds, acceptable versions are >= `lowest_acceptable_version`.
  - For exclusive bounds, acceptable versions are > `lowest_acceptable_version`.
  - If `lowest_acceptable_version` is set to `TPSC_NoBounds`, then the lowest version available (and not excluded) is acceptable.
- A single intermediate range of versions of the service that are unacceptable.
  - For inclusive bounds, excluded versions are >= `first_excluded_version`.
  - For exclusive bounds, excluded versions are > `first_excluded_version`.
  - For inclusive bounds, excluded versions are <= `last_excluded_version`.
  - For exclusive bounds, excluded versions are < `last_excluded_version`.
  - If no bounds are specified for both the first excluded version and the last excluded version, no version is excluded.
- The highest acceptable version of the service.
  - For inclusive bounds, acceptable versions are <= `highest_acceptable_version`.
  - For exclusive bounds, acceptable versions are < `highest_acceptable_version`.
  - If `highest_acceptable_version` is set to `TPSC_NoBounds`, then the highest version available (and not excluded) is acceptable.

See section 4.3.7 for the definition of the `TPSC_ServiceRange` structure which allows the caller to specify service version constraints.

3.2.3.5 tps-secure-component-type

The `tps-secure-component-type` defines the environment used to host a TPS Service. It is a UUIDv5 as defined in section 3.2.3.2 where the `name` field is set to a value uniquely identifying the type of secure component.

This specification defines the following values for the `name` field in `tps-secure-component-type`:

Table 3-1: `tps-secure-component-type` Values

Secure Component	UUID Name Field	Generated UUIDv5
GlobalPlatform compliant Trusted Execution Environment	"GPD-TEE"	`59846875-1e02-53c8-922f-5d60dd103a58`
GlobalPlatform compliant Secure Element	"GPC-SE"	`bdd658fa-44c1-5e59-b3a1-1a8f038ceb50`
Regular Execution Environment (e.g. Linux, Windows, RTOS)	"GPP-REE"	`d2dc120c-3e4a-5b1f-bece-df3825c933ae`

Other specifications MAY define further values for `tps-secure-component-type`.

3.2.3.6 tps-secure-component-instance

`tps-secure-component-instance` is used to identify a Secure Component on a Platform. It is a UUIDv5 as described in section 3.2.3.2.

This specification defines mechanisms which MAY be used for GlobalPlatform TEE and GlobalPlatform SE. Implementers MAY choose other mechanisms that produce values that are unique on a Platform.

Implementers MUST ensure that the same value is generated for `tps-secure-component-instance` each time the Platform is booted.

Privacy Note: `tps-secure-component-instance` is a privacy-sensitive identifier. Client applications need to consider privacy requirements if they plan to make `tps-secure-component-instance` available outside the Platform.

3.2.3.6.1 TEE Instances

Where the Secure Component hosting a TPS Service is a GlobalPlatform compliant TEE, the `name` field in the UUIDv5 MAY be the concatenation of:

- The UTF-8 String "GPD-TEE"
- The string representation of the value of the `gpd.tee.deviceID` property (which is itself a UUID expected to be unique).

Informative Example

```
Secure Component= "GPD-TEE"
gpd.tee.deviceID = "11567663-9fa5-4e44-9da7-174cc864cbb4"
```

`tps-secure-component-instance` is the generated UUIDv5:

```
a493ca80-f44e-5eb1-9bcd-838bce418813
```

3.2.3.6.2 Secure Element Instances

Where the Secure Component hosting a TPS Service is a GlobalPlatform compliant Secure Element, the `name` field in the UUIDv5 MAY be the concatenation of:

- The UTF-8 String "GPC-SE"
- `iin`, a UTF-8 string containing the representation in decimal of the Issuer Identification Number (see [GPCS] section 7.4.1.1).
- `cin`, a UTF-8 string containing the representation in decimal of the Card Image Number (see [GPCS] section 7.4.1.2)

Informative Example

```
Secure Component = "GPC-SE"
iin = "98268021"
cin = "38001635"
```

`tps-secure-component-instance` is the generated UUIDv5:

```
a381e1d5-6f0b-5b3f-a2fa-aa078fb00fff
```

3.2.3.7 tps-service-instance

`tps-service-instance` provides a Platform unique identifier for a TPS Service. It is a UUIDv5 as described in section 3.2.3.2.

The value of `tps-service-instance` can be used by a Connector to identify and correctly map communications to the correct service on a given Secure Component, which may host multiple services.

This specification defines mechanisms which MAY be used for GlobalPlatform compliant TEE and GlobalPlatform SE. Implementers MAY choose other mechanisms provided that the final value of `tps-service-instance` is unique on the platform.

> **Privacy Note:** `tps-service-instance` is a privacy-sensitive identifier. Client applications need to consider privacy requirements if they plan to make `tps-service-instance` available outside the Platform.

Implementers MUST ensure that the same value for `tps-service-instance` is generated after a software update that does not change `tps-service-version`, or when the Platform is rebooted.

Implementers MUST also ensure that `tps-service-instance` changes if `tps-service-version` Major Version is changed (e.g. in a software update). If the Minor Version or Patch Version change, `tps-service-instance` MUST NOT change.

3.2.3.7.1 TEE-hosted Services

Where the Secure Component hosting a TPS Service is a GlobalPlatform compliant TEE, the `name` field in the UUIDv5 MAY be the concatenation of `tps-secure-component-instance`, `ta-id`, `tps-service-name`, and `tps-service-version`.

- `tps-secure-component-instance`: Defined in section 3.2.3.6.
- `ta-id`: The UUID of the TA providing a TPS Service.
  - If TPS Service is provided as one or more TAs, `ta-id` is set to the UUID of the destination TA to which a TEE Client API ([TEE Client]) session underlying the TPS Client API session will be bound.
  - If TEE does not use a TA to provide the service, `ta-id` is Nil as defined in [RFC 4122].
  - TEEs supporting UUIDv5-based TA naming schemes SHOULD NOT use these for TAs hosting TPS Services. This ensures that the identity of a service instance remains stable if a TA receives e.g. a security update.
- `tps-service-id`: Defined in section 3.2.3.3.
- `tps-service-version`: Defined in section 3.2.3.4. Only the Major Version field is used, expressed as a hexadecimal 32-bit string with leading zeroes.

Informative Example

```
tps-secure-component-instance = "a493ca80-f44e-5eb1-9bcd-838bce418813"
ta-id = "d4e61725-1501-4bee-8dfd-dd19a81984b5"
tps-service-id = "87bae713-b08f-5e28-b9ee-4aa6e202440e"
tps-service-version = "00000002"
```

`tps-service-instance` is the generated UUIDv5:

```
9fc7dfd4-28c1-58f5-89dd-d17887a5c937
```

3.2.3.7.2 Secure Element Hosted Services

Where the Secure Component hosting a TPS Service is a GlobalPlatform compliant Secure Element, the `name` field in the UUIDv5 SHOULD be the concatenation of `tps-secure-component-instance`, `aid`, and `tps-service-version`, where:

- `tps-secure-component-instance`: Defined in section 3.2.3.6.
- `aid`: The Application Identifier of the Applet providing the TPS Service
  - If the Secure Element does not require SELECT of an AID to provide the TPS Service, aid is Nil as defined in [RFC 4122].
- `tps-service-version`: Defined in section 3.2.3.4. Only the Major Version field is used, expressed as a hexadecimal 32-bit string with leading zeroes.

Informative Example

```
tps-secure-component-instance = "a381e1d5-6f0b-5b3f-a2fa-aa078fb00fff"
aid = "DEADBEEF"
tps-service-version = "00000002"
```

`tps-service-instance` is the generated UUIDv5:

```
00f84a6a-8cb1-539f-9250-cc9c38793f1b
```

3.2.4 TPS Session

A TPS Session is an abstraction of a logical connection between a TPS Client and a TPS Service instance. The maximum number of concurrent TPS Sessions is implementation-defined, depending on the design of the TPS Service, and may depend on runtime resource constraints.

When creating a new TPS Session the Client must identify the TPS Service that it wishes to connect to by using a `tps-service-name`.

3.2.4.1 Session Login Methods

A TPS Service implementation MAY require identification or authentication of the TPS Client or the User executing it. For instance, a TPS Service implementation may restrict access to a certain set of provided services to one or more TPS Clients or Users, or identify resources hosted by the TPS Service belonging to a TPS Client and a User.

When opening a session, the TPS Client can indicate a login method it will use to identify itself. Attempting to open a session with an incorrect login method may result in a failed attempt.

3.2.5 TPS Operation

A TPS Service specifies a set of TPS Operations through which the TPS Client utilizes the TPS Service. A TPS Operation consists of one or more TPS Transactions.

3.2.6 TPS Transaction

A TPS Transaction is the unit of communication between a TPS Client and a TPS Service within a session.

- The TPS Client constructs a TPS Service request message and sends it to the TPS Service using the TPS Client API.
- The TPS Service receives the TPS Service request message via the TPS Client API, processes it, constructs a TPS Service response message, and sends the TPS Service response message to the TPS Client.
- The TPS Client receives the TPS Service response message and processes it. The TPS Client may continue by constructing a new TPS Service request message and sending it to the TPS Service in the same fashion.

The usage and content of the TPS Service request and TPS Service response messages depends on the TPS Service specification and the TPS Client's application logic.

The transaction invocation blocks the TPS Client thread, waiting for an answer from the TPS Service. A TPS Client MUST NOT use multiple threads to invoke transactions within a single TPS Session.

3.2.7 Communication Stack

The Communication stack contains required support libraries to bind the TPS Client API functionality to a particular TPS Service implementation. The Communication stack is specific to an Implementation of a particular TPS Client API and the TPS Service.

Informative Examples:

- If a TPS Service implementation is an Applet in a Secure Element, the Communication stack would contain the logic to access the Applet, including OMAPI (see [OMAPI]).
- If a TPS Service implementation is a Trusted Application in a Trusted Execution Environment, the Communication stack would contain the logic to access the Trusted Application, including the TEE Client API (defined in [TEE Client]).

3.2.8 Language Specific API and Binding

Applications and TPS Services can use a Language Specific API and a Binding to use a TPS Service.

A Language Specific API and the corresponding Binding provide an additional and optional abstraction layer on top of the TPS Client API to provide an idiomatic API for using the TPS Service from a particular programming language environment.

The Language Specific API provides a well-defined API specified using the target programming language used to develop the Application or TPS Service. The Binding provides the mapping from Language Specific API functions and function parameters to the TPS Service Protocol requests and responses and makes use of the TPS Client API to connect and communicate with the TPS Service using the TPS Service Protocol.

This document defines an optional Rust language binding in Annex A.

3.3 Usage Concepts

This section outlines some of the usage concepts underlying the TPS Client API.

3.3.1 TPSC_MessageBuffer Semantics

The `TPSC_MessageBuffer` structure manages the integrity and atomicity of operations performed between a TPS Client and a TPS Service via the TPS Client API. As such, some fields in the structure are intended to be managed via specific function invocations and should be treated as read-only from the perspective of applications using the TPS Client API.

- A `TPSC_MessageBuffer` cannot be initialized directly by an application. It MUST be initialized via a call to `TPSC_InitializeTransaction`. This ensures that any implementation-specific data is properly allocated and initialized.
- A `TPSC_MessageBuffer` cannot be finalized directly by an application. It MUST be finalized via a call to `TPSC_FinalizeTransaction`. This ensures that any implementation-specific data is properly freed.

If the Platform on which the TPS Client API executes supports multi-threading, functions that have a `TPSC_MessageBuffer` parameter MAY use it to manage reentrancy and thread safety.

3.3.2 Multi-threading

The TPS Client API is designed to support use from multiple threads concurrently, using a combination of internal thread safety within the implementation of the API, and explicit locks and serialization in the TPS Client code. TPS Client developers can assume that API functions can be used concurrently unless an exception is documented in this specification. The main exceptions are indicated below.

Note that the API can be used from multiple processes, but it may not be possible to share contexts and sessions between multiple processes due to Regular OS memory privilege separation mechanisms.

Behavior that is not thread-safe

Session structures and their corresponding lifecycle states are defined by pairs of bounding "start" and "stop" functions:

- `TPSC_OpenSession` / `TPSC_CloseSession`
- `TPSC_InitializeTransaction` / `TPSC_FinalizeTransaction`

These functions are not internally thread-safe with respect to the object being initialized or finalized. For instance, it is not valid to call `TPSC_OpenSession` concurrently using the same `TPSC_Session` structure. However, it is valid for the TPS Client to concurrently use these functions to initialize or finalize different objects; for example, two threads could initialize different `TPSC_Session` structures.

If globally shared structures need to be initialized, the TPS Client MUST use appropriate platform-specific locking schemes to ensure that the initialization of each structure occurs only once.

Once the structures described above have been initialized, it is possible to use them concurrently in other API functions, provided that the TPS Service in use supports such concurrent use.

3.3.3 Memory Layout and Management

3.3.3.1 General Principles

It is a general principle of the design of the TPS Client API that memory buffers are allocated and freed by the caller. This simple memory model reduces the likelihood of memory leaks and use-after-free errors by guaranteeing that the caller always controls memory allocation and deallocation.

3.3.3.2 Memory Management

The calling application MUST obey the following rules when managing the memory interface with the TPS Client API:

- Caller allocates and frees memory buffers.
- Caller MUST provide the correct size of an allocated memory buffer to the callee. Please take care to check whether the API requires the size to be provided in bytes, or in "number of objects of some type" that the buffer can hold.
- Caller MUST NOT move an allocated block while any other reference to it exists.
    - As an example, this can occur if an allocated block is part of a C++ vector that is resized.
- A called TPS Client API owns the contents of a buffer from the point where it is called to the point where it returns. This means in particular:
    - Caller MUST NOT mutate buffer memory until the callee has returned. On platforms where cancellation is supported, the caller only regains ownership of the buffer after the cancelled API call returns.
    - Caller MUST NOT read from a buffer which is mutated by the callee until it has returned as TPS Client API may change the contents of the memory at any time, and caller could see inconsistent memory contents.
    - Caller MAY read buffers that are not mutated by the callee.
    - TPS Client API is unaware of any synchronization primitives (semaphores, mutexes, etc.) that might be used by the caller to manage shared memory resources. It is the caller's responsibility to ensure that TPS Client API has ownership of buffer memory until the callee returns.

3.3.3.3 Structure Field Alignment

The TPS Client API will construct appropriate data structures for data within the provided buffer, including any items that are accessed via C pointers. The TPS Client API library ensures that any data structures are appropriately aligned for the caller.

Many structures contain a private `imp` field. This holds implementation-specific data whose size in memory may differ between implementations or between different versions of the same implementation. It is therefore not safe to link object code that has been compiled against different implementations or different versions of the same library. API compatibility is guaranteed only at the source code level in this version of the specification.

> **Note:** It is recommended that the TPS Client API is built with the natural structure and object alignment for the target.
>
> Implementations of the TPS Client API MUST provide information on the layout of structures so that callers can be appropriately compiled. This information MUST be present in the exported headers, and implementers are reminded that the specification of packing in the C language is compiler-dependent.
>
> The reference implementation of the TPS Client API uses the C language representation of structures *without packing directives*.

3.3.3.4 Buffer Size

Where a buffer provided by the calling application is not large enough to hold the returned data structure(s), the TPS Client API indicates this to the caller. See section 3.3.4.

The calling application is responsible for enforcing ownership semantics for the buffer. Specifically, the calling application does not access the contents of the buffer after a call to the TPS Client API until after the function call has returned.

3.3.3.5 Finalization

This specification uses the term “finalize” to describe the process of cleaning up TPS Client API resources used by a TPS Client. This specifically includes any necessary checks that the operation is legal, necessary changes to the internal state of the TPS Client API including sanitization of the contents, and freeing of allocated memory for the structures specified in the “finalize” function.

The specification of the “finalize” functions described in section 3.3.2 is stateful and requires clean TPS Client resource unwinding:

- When finalizing a `TPSC_MessageBuffer` structure, the TPS Client code MUST ensure that it is not referenced in a pending `TPSC_ExecuteTransaction` operation.
- When closing a `TPSC_Session` structure, the TPS Client code MUST ensure that there are no pending operations within the session and that all related `TPSC_MessageBuffer` structures have been finalized.
- When finalizing a `TPSC_ServiceIdentifier` structure, the TPS Client code MUST ensure that there are no pending `TPSC_OpenSession` operations pending on the structure. It can be finalized any other time, including during open sessions that were opened using the `TPSC_ServiceIdentifier` structure.

TPS Clients SHALL ensure these requirements are met, using platform-specific locking mechanisms to synchronize threads if needed. Failing to meet these obligations is a *programmer error* and may result in undefined behavior.

3.3.4 Short Buffer Handling

In this specification, memory buffers are generally defined in one of two ways:

- If the memory buffer is used only as input, a pointer (e.g. `void* buf`) holds the start of the buffer and a size parameter (e.g. `size_t size`) defines the number of entries in the buffer. This scenario is not discussed further here.
- If the memory buffer is used for both input and output, a pointer (e.g. `void* buf`) holds the start of the buffer, a size pointer (e.g. `size_t* size`) holds the size of the current contents of the buffer, and a maximum size parameter (e.g. `size_t maxsize`) holds the size of the allocated buffer (which may be larger than the current contents).

If the memory buffer provided as a parameter to a function is not large enough to contain the output from the function, handling is as follows:

The data buffer, `buf`, SHALL be allocated by the TPS Client and passed in the `buf` parameter. Because the size of the output buffer cannot generally be determined in advance, the following convention is used:

- On entry:
  - `maxsz` contains the number of bytes actually allocated in `buf`. The buffer with this number of bytes SHALL be entirely writable by the TPS Client.
  - `*sz` contains the number of bytes used by any input message in `buf`.
- On return:
  - If the output fits in the output buffer, then the Implementation SHALL write the output in `buf` and SHALL update `*sz` with the actual size of the output in bytes.
  - If the output does not fit in the output buffer, then the Implementation SHALL update `*sz` with the required number of bytes and SHALL return `TPSC_ERROR_SHORT_BUFFER`. It is implementation-dependent whether the output buffer is left untouched or contains part of the output. In any case, the TPS Client SHOULD consider that the content of the output buffer is undefined after the function returns.

If the caller sets `*sz` to `0`, then:

- The function will always return `TPSC_ERROR_SHORT_BUFFER` unless the actual output data is empty.
- The parameter `buf` can take any value, e.g. `NULL`, as it will not be accessed by the Implementation.

If the caller sets `*sz` to a non-zero value, then `buf` MUST NOT be `NULL` because the buffer starting from the `NULL` address is never writable.

3.4 Security

3.4.1 Security of the TPS Client API

The TPS Client API implementation MUST treat any input from the TPS Client as potentially malicious. TPS Services MUST assume that TPS Clients may be compromised by attack or may be purposefully malicious.

Session Login Methods

This specification defines several login methods that allow an identity token for a TPS Client to be generated by the implementation and presented to the TPS Service. This identity information is generated based on parameters controlled by some entity on the Platform, such as the OS kernel, or by a trusted entity in a Secure Component. It is a valid security model for these login tokens to be generated by a trusted process within the Platform rather than by the TPS Service itself. TPS Service developers must therefore note that the validity of this login token is bounded by the security of the Platform, not the security of the TPS Service.

3.4.2 Security of the Regular Operating System

In most implementations, the TPS Service is running in a separate Execution Environment, i.e. within a Secure Component, which exists in parallel to the Platform that runs the TPS Clients. It is important that the integration of the TPS Service alongside the Platform cannot be used to weaken the security of the Platform itself. The implementation of the TPS Service must ensure that TPS Clients cannot use the features they expose to bypass the security sandbox used by the Platform to isolate processes.

3.4.3 Security of the Communication Channel

TPS Service does not trust the TPS Client. There is no requirement to ensure confidentiality or integrity properties on the communication channel between them. TPS Services MUST treat all input as potentially malicious.

4 TPS CLIENT API

4.1 Implementation-Defined Behavior and Programmer Errors

Several functionalities within this specification are described either as *implementation-defined* or as *programmer errors*.

Implementation-Defined Behavior

When a functional behavior is described as *implementation-defined* it means that an implementation of the TPS Client API MUST consistently implement the behavior and MUST document it. However, the actual behavior is not specified as part of this specification. Application developers can choose to depend on this implementation-defined behavior but need to be aware that their code may not be portable to another Implementation.

Implementation-Defined Fields

Implementations are allowed to extend some of the data structures defined in this specification to include a single field of implementation-defined type, named `imp`. Implementations MUST NOT add new fields outside of `imp`. The size of the `imp` field MUST be known at compile time.

The implementation can use the `imp` field to hold any private data that it wants to attach to the structure, and clients of the TPS Client API MUST NOT directly access the contents of the `imp` field.

Programmer Error

This specification identifies errors that can only occur due to mistakes by the programmer. They are triggered through incorrect use of the API by a program rather than by run-time errors such as out-of-memory conditions.

The Implementation is not required to gracefully handle programmer errors, or even to behave consistently, but MAY choose to generate a programmer-visible response. This response could include a failing assertion, an informative return code if the function can return one, a diagnostic log file, etc. In the event of a programmer error, the Implementation MUST ensure the stability and security of the TPS Service and the shared communication subsystem in the Regular OS environment, because these modules are shared amongst all Applications and MUST NOT be affected by the misbehavior of a single Application.

4.2 Header File

The header file for the TPS Client API SHALL have the name `"tpsc_client_api.h"`.

```
#include "tpsc_client_api.h"
```

4.3 Data Types

4.3.1 Basic Types

This specification makes use of the integer and Boolean C types as defined in the C99 standard (ISO/IEC 9899:1999 – [C99]). In the event of any difference between the definitions in this specification and those in [C99], C99 shall prevail. The following standard C types are used:

• `uint32_t`:	a 32-bit unsigned integer
• `uint16_t`:	a 16-bit unsigned integer
• `uint8_t`:	an 8-bit unsigned integer
• `char`:	a character
• `size_t`:	an unsigned integer large enough to hold the size of an object in memory

4.3.2 TPSC_ConnectionData

Since: TPS Client API v1.0

```
#include <sys/types.h>

typedef enum {
    TPSC_NoConnectionData,
    TPSC_GID,
    TPSC_Proprietary
} TPSC_ConnectionData_Tag;

typedef struct {
    TPSC_ConnectionData_Tag tag;
    union {
        gid_t gid;
        const void *proprietary;
    };
} TPSC_ConnectionData;
```

Description

Note: In this version of the specification, `TPSC_ConnectionData` data fields are defined for Unix-based platforms and platforms that can emulate Unix group and process behavior.

The definition for `gid_t` used above is found in `sys/types.h` on Unix systems.

The `TPSC_ConnectionData` structure allows a caller to provide any data required to authorize a connection to a Secure Component, with content that depends on the Session Login Method used (see section 4.4.2). It consists of the following fields:

- `tag` is set to a value which distinguishes the type of any additional information required to authorize the connection, which is provided by one of the options in the union. At most, one of the union fields is set. When `tag` is `TPSC_NoConnectionData`, the contents of the union are ignored by the callee and SHOULD NOT be set by the caller.
- `gid` is set by the caller, and considered valid by the callee when the tag field is `TPSC_GID`. It is set to the value of a Unix group ID.

- `proprietary` is set by the caller and considered valid by the callee when the tag field is `TPSC_Proprietary`. It is set to point to a value that is understood by the callee.

Table 4-1 defines how `TPSC_ConnectionData` is used for different values of Session Login Method (as defined in section 4.4.2).

Table 4-1: `TPSC_ConnectionData` for Session Login Methods

Session Login Method	`TPSC_ConnectionData`

`TPSC_LOGIN_PUBLIC`
`TPSC_LOGIN_USER`
`TPSC_LOGIN_APPLICATION`
`TPSC_LOGIN_USER_APPLICATION`
	`TPSC_ConnectionData.tag` field is set to `TPSC_CONNECTIONDATA_NONE`.
	`TPSC_ConnectionData` union fields are all ignored.

`TPSC_LOGIN_GROUP` `TPSC_LOGIN_GROUP_APPLICATION`	`TPSC_ConnectionData.tag` field is set to `TPSC_CONNECTIONDATA_GID`. The value in `TPSC_ConnectionData.gid` field is set to the Group ID that this TPS Client wants to connect as.
Any reserved value from Table 4-3	`TPSC_ConnectionData.tag` field is set to `TPSC_CONNECTIONDATA_PROPRIETARY`. The value in `TPSC_ConnectionData.proprietary` MAY be set to an implementation-defined value.

Note: The API intentionally omits any form of support for static login credentials, such as PIN or password entry. The login methods supported in the API are only those that have been identified as requiring support by the Platform.

4.3.3 TPSC_MessageBuffer

Since: TPS Client API v1.0

```
typedef struct
{
    uint8_t*                        message;
    size_t                          size;
    const size_t                    maxsize;
    const TPSC_MessageBufferPriv    imp;
} TPSC_MessageBuffer;
```

Description

This type is used as a container for TPS Service request and response messages.

The fields of this structure have the following meaning:

- `message` is a pointer to the first byte of a region of memory, i.e. a message buffer, of length `maxsize`, which can contain a TPS Service request or response message.
- `size` is the size of the current `message`, in bytes. When an operation completes, the Implementation must update this field to reflect the actual or required size of the output.
    - When the TPS Client has written the request message in the `message` field, then it MUST update the `size` field with the actual size of the request message in bytes.
    - When the Implementation has written the response message in the `message` field, then it MUST update the `size` field with the actual size of the response message in bytes.
    - If the maximum size of the `message` field was not large enough to contain the whole response message, the Implementation MUST update the `size` field with the size of the message buffer requested by the TPS Service.
- `maxsize` is the size of the referenced memory region, in bytes, denoting the maximum size for the message.
- `imp` contains any additional implementation-defined data structure of type `TPSC_MessageBufferPriv` attached to the `TPSC_MessageBuffer` structure.
    - `imp` MUST contain any data fields necessary to allow an implementation of the TPS Client API to support the usage concepts defined in section 3.3.
    - Clients of the TPS Client API MUST NOT access this field.

4.3.4 TPSC_Result

Since: TPS Client API v1.0

```
Typedef uint32_t TPSC_Result;
```

This type is used to contain return codes that are the results of invoking TPS Client API functions. See section 4.4.1 for a list of return codes defined by this specification.

4.3.5 TPSC_ServiceBound

Since: TPS Client API v1.0

```
typedef enum {
    TPSC_Inclusive,
    TPSC_Exclusive,
    TPSC_NoBounds
} TPSC_ServiceBound_Tag;

typedef struct {
    TPSC_ServiceBound_Tag tag;
    union {
        struct {
            TPSC_ServiceVersion inclusive;
        };
        struct {
            TPSC_ServiceVersion exclusive;
        };
    };
} TPSC_ServiceBound;
```

Description

This type allows specification of service version bounds. It is used only in the context of a `TPS_ServiceRange` (see section 4.3.7)

The fields of this structure have the following meaning:

- `tag` is set to one of the following (see section 3.2.3.4.4 for a detailed description of inclusive and exclusive version range behavior):
    - `TPSC_Inclusive` to indicate that the service version bound specified by this instance of `TPSC_ServiceBound` is inclusive.
    - `TPSC_Exclusive` to indicate that the service version bound specified by this instance of `TPSC_ServiceBound` is exclusive.
    - `TPSC_NoBounds` to indicate that no service version bound is specified.
- The contents of the union define the service version as follows:
    - `inclusive` is set to a `TPSC_ServiceVersion` value indicating inclusive version bounds when `tag` is `TPSC_Inclusive`.
    - `exclusive` is set to a `TPSC_ServiceVersion` value indicating exclusive version bounds when `tag` is `TPSC_Exclusive`.
    - No union field is set when `tag` is `TPSC_NoBounds`, and the callee will ignore any value.

4.3.6 TPSC_ServiceIdentifier

Since: TPS Client API v1.0

```
typedef struct
{
  TPSC_UUID                     service_instance;
  TPSC_UUID                     service_id;
  TPSC_UUID                     secure_component_type;
  TPSC_UUID                     secure_component_instance;
  TPSC_ServiceVersion           service_version;
} TPSC_ServiceIdentifier;
```

Description

This type denotes a TPS Service instance, the logical container identifying a particular TPS Service implementation on the Platform.

The fields of this structure have the following meaning:

- `service_instance` is a `TPSC_UUID` that uniquely distinguishes a particular TPS Service on a given Platform. See section 3.2.3.7.
- `service_id` is a `TPSC_UUID` that identifies the TPS Service being provided. See section 3.2.3.3.
- `secure_component_type` is a `TPSC_UUID` that identifies the type of Secure Component providing a TPS Service. See section 3.2.3.5.
- `secure_component_instance` is a `TPSC_UUID` that distinguishes a particular Secure Component providing a TPS Service. See section 3.2.3.6.
- `service_version` is a `TPSC_ServiceVersion` indicating the version of the TPS Service identified by this `TPSC_ServiceIdentifier`.

4.3.7 TPSC_ServiceRange

Since: TPS Client API v1.0

```
typedef struct {
  TPSC_ServiceBound  lowest_acceptable_version;
  TPSC_ServiceBound  first_excluded_version;
  TPSC_ServiceBound  last_excluded_version;
  TPSC_ServiceBound  highest_acceptable_version;
} TPSC_ServiceRange;
```

Description

`TPSC_ServiceRange` allows a caller to specify which versions of a TPS Service implementation are acceptable to it, allowing version constraints to be used in the service discovery process. This is described in more detail in section 3.2.3.4.4.

`TPSC_ServiceRange` consists of four values which allow the caller to specify the lowest and highest acceptable versions of a TPS Service to be specified, as well as permitting a specific set of service versions to be excluded, should a need for this arise.

- `lowest_acceptable_version`: Specifies the lowest acceptable version of a service implementation to be returned in service discovery.
- `first_excluded_version`: Specifies the lowest version to be excluded from the service implementations returned in service discovery.
- `last_excluded_version`: Specifies the highest version to be excluded from the service implementations returned in service discovery.
- `highest_acceptable_version`: Specifies the highest acceptable version of a service implementation to be returned in service discovery.

4.3.8 TPSC_ServiceSelector

Since: TPS Client API v1.0

```
typedef struct {
  TPSC_UUID           service_id;
  TPSC_UUID           secure_component_type;
  TPSC_UUID           secure_component_instance;
  TPSC_ServiceRange   service_version_range;
} TPSC_ServiceSelector;
```

Description

The `TPSC_ServiceSelector` structure is populated prior to calling the `TPSC_DiscoverServices` function. It specifies to `TPSC_DiscoverServices` which services the caller wants included in the returned list of services.

The structure members are used to filter from the full set of TPS Services available on a Platform as follows:

- `service_id`
  - If this is a valid UUID, the returned list of services includes only services with this UUID as their `tps-service-id`.
  - If this is `TPSC_UUID_NIL`, the returned list of services matches any TPS Service.
- `secure_component_type`
  - If this is a valid UUID, the returned list of services includes only services hosted by Secure Components with a matching `tps-secure-component-type`.
  - If this is `TPSC_UUID_NIL`, the returned list of services will include those hosted by any type of Secure Component.
- `secure_component_instance`
  - If this is a valid UUID, the returned list of services includes only services hosted by a Secure Component with `tps-secure-component-instance` matching the value provided.
  - If this is `TPSC_UUID_NIL`, the returned list of services will include those hosted by any Secure Component instance.
- `service_version_range`
  - A `TPSC_ServiceRange` instance containing a version range specification as described in section 3.2.3.4.4. The returned list of services will contain only services where `tps-service-version` matches the range specification.
    - If the caller does not care about the service version range, the fields of `TPSC_ServiceRange` can all be set to `TPSC_NoBounds`.

4.3.9 TPSC_ServiceVersion

Since: TPS Client API v1.0

```
typedef struct TPSC_ServiceVersion {
  uint32_t   major_version;
  uint32_t   minor_version;
  uint32_t   patch_version;
};
```

Description

This type denotes a `tps-service-version`. See section 3.2.3.4.

- `TPSC_ServiceVersion.major_version` holds the Major Version of the TPS Service.
- `TPSC_ServiceVersion.minor_version` holds the Minor Version of the TPS Service.
- `TPSC_ServiceVersion.patch_version` holds the Patch Version of the TPS Service.

4.3.10 TPSC_Session

Since: TPS Client API v1.0

```
typedef struct
{
   const TPSC_UUID*           const service_id;
   uint32_t                   session_id;
   const TPSC_SessionPriv     imp;
} TPSC_Session;
```

Description

This type denotes a TPS Service session, the logical container linking a TPS Client and a particular TPS Service implementation.

The fields of this structure have the following meaning:

- `service_id` is a pointer to a `TPSC_UUID` that is a `tps-service-id`. This is associated with the `TPSC_Session`.
- `session_id` uniquely identifies the session.
- `imp` contains any additional implementation-defined data structure of type `TPSC_SessionPriv` attached to the `TPSC_Session` structure.
  - `imp` MUST contain any data fields necessary to allow an implementation of the TPS Client API to support the usage concepts defined in section 3.3.
  - Clients of the TPS Client API MUST NOT access this field.

4.3.11 TPSC_UUID

Since: TPS Client API v1.0

```
typedef struct
{
   uint8_t bytes[16];
} TPSC_UUID;
```

Description

This type is used to encapsulate a UUID.

The fields of this structure have the following meaning:

- `bytes` is an array of 16 x `uint8_t` which represents a UUID encoded as bytes.

Informative Example

If the string representation of the UUID of a `tps-service-id` is `720eeb3d-058d-5bdf-80d0-958c74f6de57`, then the corresponding `TPSC_UUID` can be initialized as follows:

```
TPSC_UUID example = {
  .bytes = { 0x72, 0x0e, 0xeb, 0x3d, 0x05, 0x8d, 0x5b, 0xdf,
             0x80, 0xd0, 0x95, 0x8c, 0x74, 0xf6, 0xde, 0x57 }
};
```

4.4 Constants

4.4.1 Return Codes

The following function return codes, of type `TPSC_Result` (see section 4.3.4), are defined by this specification.

Table 4-2: API Return Code Constants

Name	Value	Description / Cause
`TPSC_SUCCESS`	`0x00000000`	The operation was successful.
`TPSC_ERROR_GENERIC`	`0xF0090000`	Non-specific cause.
`TPSC_ERROR_ACCESS_DENIED`	`0xF0090001`	Access privileges are not sufficient.
`TPSC_ERROR_CANCEL`	`0xF0090002`	The operation was cancelled.
`TPSC_ERROR_BAD_FORMAT`	`0xF0090003`	Input data was of invalid format.
`TPSC_ERROR_NOT_IMPLEMENTED`	`0xF0090004`	The requested operation should exist but is not yet implemented. See note following table.
`TPSC_ERROR_NOT_SUPPORTED`	`0xF0090005`	The requested operation is valid but is not supported in this implementation.
`TPSC_ERROR_NO_DATA`	`0xF0090006`	Expected data was missing.
`TPSC_ERROR_OUT_OF_MEMORY`	`0xF0090007`	System ran out of resources.
`TPSC_ERROR_BUSY`	`0xF0090008`	The system is busy working on something else.
`TPSC_ERROR_COMMUNICATION`	`0xF0090009`	Communication with a remote party failed.
`TPSC_ERROR_SECURITY`	`0xF009000A`	A security fault was detected.
`TPSC_ERROR_SHORT_BUFFER`	`0xF009000B`	The supplied buffer is too short for the generated output.
`TPSC_ERROR_DEPRECATED`	`0xF009000C`	A warning that the called function is deprecated. The implementation is assumed to have returned a correct result when this value is set.
`TPSC_ERROR_BAD_IDENTIFIER`	`0xF009000D`	A supplied UUID was not recognized for the requested usage.
`TPSC_ERROR_NULL_POINTER`	`0xF009000E`	A pointer value passed was `NULL`.
`TPSC_ERROR_BAD_STATE`	`0xF009000F`	A transaction was incorrectly initialized or was returned in an incorrect state.
`TPSC_ERROR_TIMEOUT`	`0xF0090010`	A timeout occurred when waiting for some action to complete.
`TPSC_ERROR_PLATFORM`	`0xF0090011`	An unrecoverable error was reported by the platform.
`TPSC_ERROR_RUNTIME_ERROR`	`0xF0090012`	A runtime error was reported by the platform.
Implementation-Defined	`0xF0000001 – 0xF000FFFE`	
Reserved for Future Use	All other values	

Note: Production implementations of the TPS Client API SHOULD NOT return `TPSC_ERROR_NOT_IMPLEMENTED`. It is intended for use by implementers of the TPS Client API during development. To denote non-implementation of an optional feature, implementations SHOULD return `TPSC_ERROR_NOT_SUPPORTED`.

4.4.2 Session Login Methods

The following constants, of type `uint32_t`, are defined by this specification. These are used to indicate which of the Application's identity credentials the implementation will use to determine access control permission to functionality provided by, or keys stored by, the TPS Service.

Session Login Methods are designed to be orthogonal from each other, in accordance with the identity token(s) defined for each constant. For example, the credentials generated for `TPSC_LOGIN_APPLICATION` MUST depend only on the identity of the TPS Client, and not on the user running it. If two users use the same TPS Client, the Implementation MUST assign the same login identity to both users so that they can access the same assets held inside the TPS Service. These identity tokens MUST be persistent within one Implementation, across multiple invocations of the application and across power cycles, enabling them to be used to disambiguate persistent storage.

Note that this specification does not guarantee separation based on use of different login methods. In many embedded platforms there is no notation of "group" or "user" so these login methods may fall back to `TPSC_LOGIN_PUBLIC`. Details of generating the credential for each login method are implementation-defined.

Table 4-3: Session Login Methods

Name	Value	Comment
`TPSC_LOGIN_PUBLIC`	`0x00000000`	No login data is provided.
`TPSC_LOGIN_USER`	`0x00000001`	The Platform provides login data about the user running the Application process.
`TPSC_LOGIN_GROUP`	`0x00000002`	The Platform provides login data about the group running the Application process.
`TPSC_LOGIN_APPLICATION`	`0x00000003`	The Platform provides login data about the running Application itself.
`TPSC_LOGIN_USER_APPLICATION`	`0x00000004`	The Platform provides login data about the user running the Application and about the Application itself.
`TPSC_LOGIN_GROUP_APPLICATION`	`0x00000005`	The Platform provides login data about the group running the Application and about the Application itself.
`TPSC_LOGIN_ILLEGAL_VALUE`	`0x7FFFFFFF`	This value MUST NOT be used by application programmers. It is reserved for functional compliance use.
Reserved for Implementation-Defined login methods.	`0x80000000 – 0xFFFFFFFF`	Behavior is implementation-defined.
All other constant values are Reserved for Future Use.		

4.4.3 TPSC_UUID_NIL

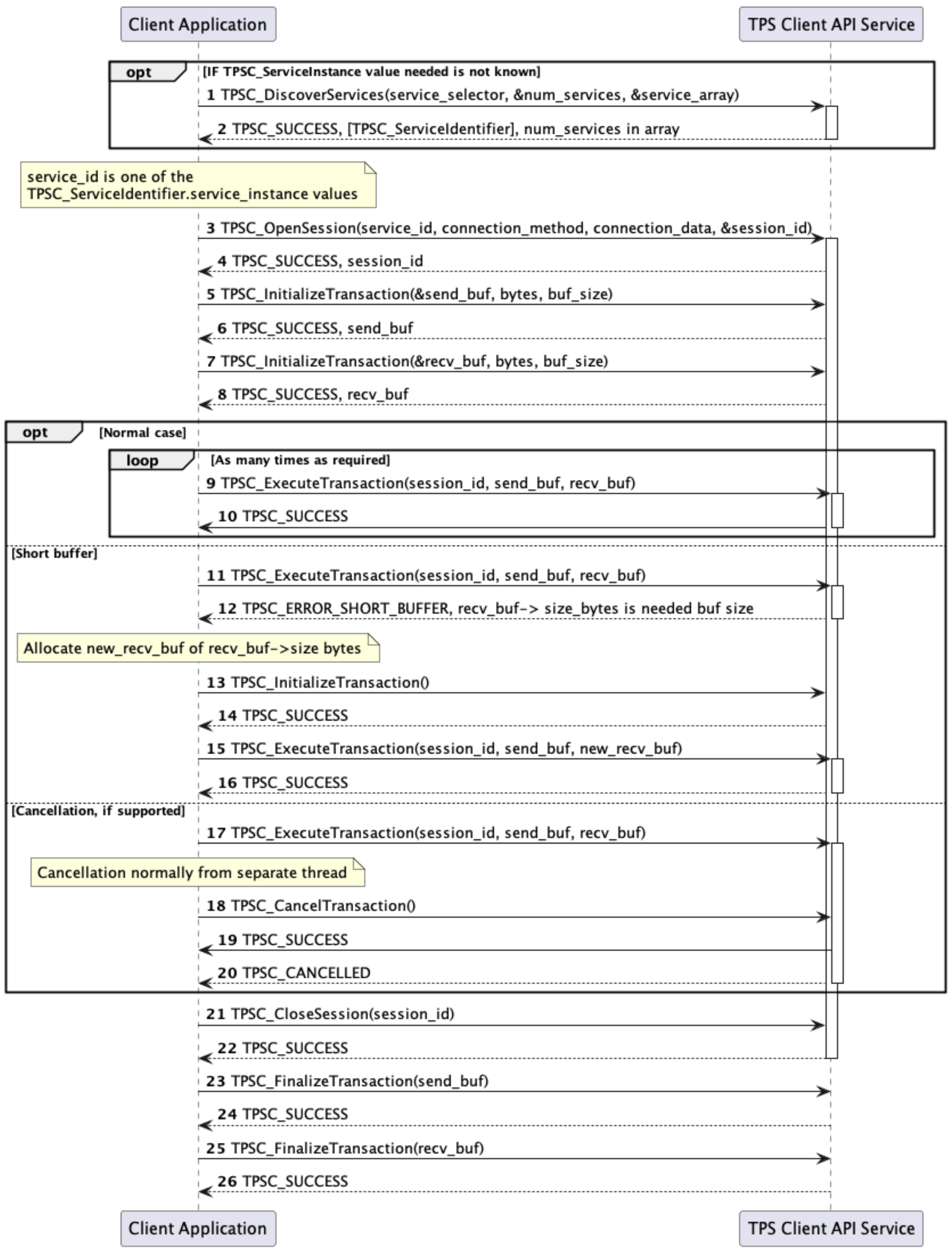
The Nil UUID, as defined in [RFC 4122] section 4.1.7

```
#define TPSC_UUID_NIL { \
    .bytes = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }\
}
```

4.5 Functions

The following sub-sections specify the behavior of the functions within the TPS Client API. Figure 4-1 shows a highly simplified outline of how these functions might be called in a real application.

Figure 4-1: Typical Call Sequence


Client Application
TPS Client API Service
opt
[IF TPSC_ServiceInstance value needed is not known]
1 TPSC_DiscoverServices(service_selector, &num_services, &service_array)
2 TPSC_SUCCESS, [TPSC_ServiceIdentifier], num_services in array
service_id is one of the TPSC_ServiceIdentifier.service_instance values
3 TPSC_OpenSession(service_id, connection_method, connection_data, &session_id)
4 TPSC_SUCCESS, session_id
5 TPSC_InitializeTransaction(&send_buf, bytes, buf_size)
6 TPSC_SUCCESS, send_buf
7 TPSC_InitializeTransaction(&recv_buf, bytes, buf_size)
8 TPSC_SUCCESS, recv_buf
opt
[Normal case]
loop
[As many times as required]
9 TPSC_ExecuteTransaction(session_id, send_buf, recv_buf)
10 TPSC_SUCCESS
[Short buffer]
11 TPSC_ExecuteTransaction(session_id, send_buf, recv_buf)
12 TPSC_ERROR_SHORT_BUFFER, recv_buf-> size_bytes is needed buf size
Allocate new_recv_buf of recv_buf->size bytes
13 TPSC_InitializeTransaction()
14 TPSC_SUCCESS
15 TPSC_ExecuteTransaction(session_id, send_buf, new_recv_buf)
16 TPSC_SUCCESS
[Cancellation, if supported]
17 TPSC_ExecuteTransaction(session_id, send_buf, recv_buf)
Cancellation normally from separate thread
18 TPSC_CancelTransaction()
19 TPSC_SUCCESS
20 TPSC_CANCELLED
21 TPSC_CloseSession(session_id)
22 TPSC_SUCCESS
23 TPSC_FinalizeTransaction(send_buf)
24 TPSC_SUCCESS
25 TPSC_FinalizeTransaction(recv_buf)
26 TPSC_SUCCESS
Client Application
TPS Client API Service

4.5.1 Documentation Format

Since: TPS Client API version that first defined this function

```
Function Prototype
```

Description

This topic describes the behavior of the function.

Parameters

This topic describes each of the function parameters.

Return

This topic lists the possible return values. Note that this list is not comprehensive, and often leaves some choice over error return codes to the Implementation. However, if restrictions do exist, then this topic will document them.

Programmer Error

This topic documents cases of *programmer error* – error cases that MAY be detected by the Implementation, but that MAY also perform in an unpredictable manner. This topic is not exhaustive and does not document cases such as passing an invalid pointer or a `NULL` pointer where the body text states that the pointer must point to a valid structure.

Implementer Notes

This topic highlights key points about the intended use of the function.

4.5.2 TPSC_CancelTransaction

Since: TPS Client API v1.0

```
TPSC_Result TPSC_CancelTransaction (
   TPSC_MessageBuffer*       transaction
);
```

Description

Note: The implementation MUST maintain an association of each `TPSC_MessageBuffer` instance to any ongoing transaction of which it is a part, and this association determines which transaction to cancel. See section 4.5.4.

The function requests the cancellation of a pending transaction invocation operation. As this is a synchronous API, this function must be called from a thread other than the one executing the `TPSC_OpenSession` or `TPSC_ExecuteTransaction` function.

This function just sends a cancellation signal to the TPS Client API and returns immediately; the operation is not guaranteed to have been cancelled when this function returns. In addition, the cancellation request is just a hint; the TPS Client API or the TPS Service MAY ignore the cancellation request.

It is valid to call this function using a `TPSC_MessageBuffer` structure any time after the TPS Client has called `TPSC_ExecuteTransaction`. A `TPSC_CancelTransaction` can be requested on a transaction before it is invoked, during invocation, and after invocation.

TPS Clients MUST NOT reuse the `TPSC_MessageBuffer` structure for another transaction until the cancelled transaction has returned in the thread executing the `TPSC_OpenSession` or `TPSC_ExecuteTransaction` function.

If `TPSC_CancelTransaction` is called with `transaction` set to `NULL`, the implementation MUST return `TPSC_ERROR_CANCEL` if the implementation supports cancellation or `TPSC_ERROR_NOT_SUPPORTED` if cancellation is not supported. This mechanism can be used by a TPS Client to determine whether an implementation supports cancellation.

In many cases it will be necessary for the TPS Client to detect whether the transaction was cancelled, or whether it completed normally. If the transaction was cancelled, the return code of the `TPSC_OpenSession` or `TPSC_ExecuteTransaction` function MUST be `TPSC_ERROR_CANCEL`.

Parameters

- `transaction`: A pointer to a TPS Client instantiated `TPSC_MessageBuffer` structure, or `NULL`.

Return

- `TPSC_SUCCESS`: `transaction` was not `NULL` and the implementation received the cancellation request.
- `TPSC_ERROR_CANCEL`: `transaction` was `NULL` and the TPS Client API implementation supports cancellation.
- `TPSC_ERROR_NOT_SUPPORTED`: The TPS Client API implementation does not support cancellation.
- `TPSC_ERROR_NULL_POINTER`: `transaction` was `NULL`.
- `TPSC_ERROR_GENERIC`: Any other error.

Programmer Error

None

Implementer Notes

None

4.5.3 TPSC_CloseSession

Since: TPS Client API v1.0

```
TPSC_Result TPSC_CloseSession(
   TPSC_Session*          session
);
```

Description

The function closes a session that was opened with a TPS Service.

All transactions within the session MUST have completed before calling this function.

The Implementation MUST do nothing if the `session` parameter is `NULL`.

Parameters

- `session`: The session to close.

Return

- `TPSC_SUCCESS`: Session closed successfully.
- `TPSC_ERROR_COMMUNICATION`: No instance of the Connector to which this session refers can be found. This could occur because the Secure Component to which a Connector is associated has been removed.
- `TPSC_ERROR_NULL_POINTER`: `session` is `NULL`.
- `TPSC_ERROR_BADSTATE`: The session state information is incorrect or corrupted.

Programmer Error

The following usage of the API is a programmer error:

- Calling with a `session` that still has transactions running.
- Attempting to close the same session concurrently from multiple threads.
- Attempting to close the same session more than once.

Implementer Notes

None

4.5.4 TPSC_DiscoverServices

Since: TPS Client API v1.0

```
TPSC_Result TPSC_DiscoverServices (
   const TPSC_ServiceSelector* const   service_selector,
   TPSC_ServiceIdentifier* const       service_array,
   size_t* const                       num_services
);
```

Description

The function discovers all TPS Services available via the TPS Client API that match the `service_selector` criteria.

The Implementation MUST assume that on entry, all fields of the `service_array` structure are in an undefined state. When this function returns `TPSC_SUCCESS`, the Implementation MUST have populated this structure with any information necessary for subsequent operations within the `TPSC_ServiceIdentifier array` structure.

The caller is responsible for ensuring that `service_array` is appropriately aligned to contain instances of `TPSC_ServiceIdentifier`.

Parameters

- `service_selector`: A pointer to an instance of a `TPSC_ServiceSelector` structure which specifies the search parameters to be used when populating the returned array of `TPSC_ServiceIdentifier`.
- `service_array`: A pointer to a contiguously allocated memory block of at least `(sizeof(TPSC_ServiceIdentifier) * (*num_services))` bytes which will be used to hold `TPSC_ServiceIdentifier` structures. On return, this will contain an array of `TPSC_ServiceIdentifier` structures that identify the list of TPS Services that are available and match the selector criteria.
- `num_services`: On entry, a pointer to an integer that indicates the number of instances of `TPSC_ServiceIdentifier` that `service_array` can hold. On return, points to the number of items in the list. If `TPSC_ERROR_SHORT_BUFFER` is returned, the value pointed to by `num_services` indicates the number of service items that `service_array` needs to hold for a successful return.

Return

- `TPSC_SUCCESS`: Discovery was successful.
- `TPSC_ERROR_BAD_FORMAT`: `service_selector` was not valid.
- `TPSC_ERROR_COMMUNICATION`: Failed to establish communication with the Secure Component(s) implementing the service.
- `TPSC_ERROR_NULL_POINTER`: One or more of the pointer values passed were `NULL`.
- `TPSC_ERROR_SHORT_BUFFER`: Provided `service_array` was not large enough to hold the `TPSC_ServiceIdentifier` array.
- `TPSC_ERROR_GENERIC`: Any other error.

Programmer Error

None

Implementer Notes

`TPSC_DiscoverServices` MUST be reentrant and thread-safe on Platforms permitting such an implementation.

4.5.5 TPSC_ExecuteTransaction

Since: TPS Client API v1.0

```
TPSC_Result TPSC_ExecuteTransaction(
   const TPSC_Session*           session,
   const TPSC_MessageBuffer*     send_buf,
         TPSC_MessageBuffer*     recv_buf)
```

Description

The function sends a request message and receives a response message within the context of the specified session.

The parameter `session` MUST point to a valid open session.

Transaction Handling

A transaction MUST carry a transaction payload. The parameters `send_buf` and `recv_buf` MUST point to `TPSC_MessageBuffer` structures previously initialized by the TPS Client.

The `send_buf` and `recv_buf` structures contain state information that is used to manage cancellation of the transaction and may be shared with other threads.

The transaction payload is handled by sequentially executing the following steps:

1. TPS Client has initialized the `TPSC_MessageBuffer` structures by using the `TPSC_InitializeTransaction` function.
2. TPS Client has prepared a TPS Service request message.
3. TPS Client populates the `send_buf` structure with the TPS Service request message after which the `message` field contains the TPS Service request message and the `size` field contains the size of the TPS Service request message in bytes.
4. TPS Client invokes the `TPSC_ExecuteTransaction` function with the `send_buf` and `recv_buf` parameters. If the Implementation supports cancellation, internal state information managing cancellation MUST be set to indicate that a transaction is in progress.
5. The `send_buf` contents are sent to the TPS Service. During the execution of the transaction, the TPS Service reads the TPS Service request message held in the `message` field of the `send_buf`, executes the request and creates a TPS Service response message, populates the `message` field of the `recv_buf` to contain the TPS Service response message, and updates the `size` parameter of the `recv_buf` to indicate the size of the TPS Service response message.
6. When the TPS Service completes the transaction, control is passed back to the calling TPS Client code. When the transaction is complete, internal state information managing cancellation, if supported, MUST be set to indicate that there is no transaction in progress.

Parameters

- `session`: The open session in which the transaction will be invoked.
- `send_buf`: A pointer to a TPS Client initialized `TPSC_MessageBuffer` structure holding the message to send.
- `recv_buf`: A pointer to a TPS Client initialized `TPSC_MessageBuffer` structure which will hold the returned data.

Return

- `TPSC_SUCCESS`: Transaction was successfully executed.
- `TPSC_ERROR_NO_DATA`: `send_buf`, `recv_buf`, or `session` is `NULL`.
- `TPSC_ERROR_BAD_FORMAT`: `send_buf` or `recv_buf` was not initialized before the function was called.
- `TPSC_ERROR_SHORT_BUFFER`: The buffer allocated in `recv_buf` is not large enough to contain the response. In this case, the handling in section 3.3.4 applies and `recv_buf->size` contains the size of the buffer required to hold the TPS Service response message.

Programmer Error

The following usage of the API is a programmer error:

- Calling with a `session` that is not an open session.
- Using the same `session` concurrently for multiple operations.
- Calling with invalid content in the `message` field of the `TPSC_MessageBuffer` structure.
- Using the same `TPSC_MessageBuffer` structure on different threads.

Implementer Notes

None

4.5.6 TPSC_FinalizeTransaction

Since: TPS Client API v1.0

```
TPSC_Result TPSC_FinalizeTransaction(
   TPSC_MessageBuffer* const     transaction
);
```

Description

The function finalizes a `transaction` structure, allowing the `transaction->message` buffer to be safely freed by the caller.

Parameters

- `transaction`: A previously initialized `TPSC_MessageBuffer` instance.

Return

The following values can be returned.

- `TPSC_SUCCESS`: `transaction` was successfully finalized.
- `TPSC_ERROR_BAD_STATE`: `transaction` was not correctly initialized.
- `TPSC_ERROR_NULL_POINTER`: `transaction` was `NULL`.
- `TPSC_ERROR_GENERIC`: Any other error.

Programmer Error

It is an error to call `TPSC_FinalizeTransaction` on a `TPSC_MessageBuffer` that is still owned by an ongoing transaction.

Implementer Notes

It is strongly recommended that the contents of `transaction->message` are cleared as part of this function call.

4.5.7 TPSC_InitializeTransaction

Since: TPS Client API v1.0

```
TPSC_Result TPSC_InitializeTransaction(
   TPSC_MessageBuffer* const     transaction,
   uint8_t* const                buffer,
   const size_t                  buf_size);
```

Description

The function initializes a `transaction` structure for use in the `TPSC_ExecuteTransaction` function. The `transaction` structure may be used multiple times with the `TPSC_ExecuteTransaction` function.

The Implementation MUST assume that on entry, all fields of this `transaction` structure are in an undefined state. When this function returns `TPSC_SUCCESS`, the Implementation MUST have populated the `transaction` structure with any information necessary for subsequent operations within the `transaction` structure.

Parameters

- `transaction`: If the function returns `TPSC_SUCCESS`, the parameters of `transaction` are updated as follows:
  - `message` is set to `buffer`. This implies that ownership of the buffer has passed to the `TPSC_ExecuteTransaction` instance and this ownership is released only through a call to `TPSC_FinalizeTransaction`.
  - `size` is set to zero.
  - `maxsize` indicates the maximum size of message that can be stored in the `TPSC_MessageBuffer`.
- `buffer`: A pointer to a buffer containing `buf_size` bytes which will be used to contain the `TPSC_MessageBuffer` structure after its initialization. The caller MUST ensure that the start address of buffer is appropriately aligned to hold any structure type.
- `buf_size`: The size, in bytes, of the buffer that will be used to construct the `TPSC_MessageBuffer` instance.

Return

- `TPSC_SUCCESS`: `transaction` was successfully initialized.
- `TPSC_ERROR_BAD_STATE`: `transaction` was already initialized before the function was called.
- `TPSC_ERROR_NULL_POINTER`: `transaction` was `NULL`.
- `TPSC_ERROR_GENERIC`: Any other error.

Programmer Error

The following usage of the API is a programmer error:

- Attempting to initialize the same `transaction` structure concurrently from multiple threads.
- Attempting to initialize the same `transaction` structure more than once.
- Attempting to directly free buffer before a call to `TPSC_FinalizeTransaction`.

Implementer Notes

None

4.5.8 TPSC_OpenSession

Since: TPS Client API v1.0

```
TPSC_Result TPSC_OpenSession(
   const TPSC_UUID* const                service,
   const uint32_t                        login_method,
   const TPSC_ConnectionData* const      connection_data,
         TPSC_Session* const             session
);
```

Description

The function opens a new session between the TPS Client and the TPS Service identified by the `service` structure.

The Implementation MUST assume that on entry, all fields of the `session` structure are in an undefined state. When this function returns `TPSC_SUCCESS`, the Implementation MUST have populated this structure with any information necessary for subsequent operations within the session.

The target TPS Service is identified by the `TPS_UUID` instance passed in the parameter `service`.

The session MAY be opened using a specific login method that can carry additional connection data, such as data about the user or user-group running the TPS Client, or about the TPS Client itself. This allows the TPS Service to implement access control methods that separate functionality or data accesses for different actors.

Standard login methods are defined in section 4.4.2 but implementation-defined login methods MAY also be defined.

Note: The API intentionally omits any form of support for static login credentials, such as PIN or password entry. The login methods supported in the API are only those that have been identified as requiring support by the Platform.

Parameters

- `service`: A pointer to a `TPS_UUID` structure that uniquely identifies the TPS Service to connect to – a value that was returned as a `TPSC_ServiceIdentifier.service_instance`. This parameter cannot be set to `NULL`.
- `login_method`: The method of login to use. Refer to section 4.4.2 for more details.
- `connection_data`: Any necessary data required to support the login method chosen.
- `session`: A pointer to a `TPSC_Session` structure that identifies the session. Session structure must be uninitialized.

Return

- `TPSC_SUCCESS`: Session was successfully opened.
- `TPSC_ERROR_BAD_IDENTIFIER`: The value provided for service does not identify a `tps-service-instance` on this platform (see section 3.2.3.7).
- `TPSC_ERROR_BUSY`: The requested operation failed because the system was busy. This can occur when the limit of supported sessions is reached.
- Another error code from Table 4-2: Opening the session was not successful.

Programmer Error

The following usage of the API is a programmer error:

- Calling with `connection_data` set to `NULL` if connection data is required by the specified login method.
- Calling with `service` or `session` set to `NULL` or pointing to an unallocated memory area.
- Attempting to open a session using the same `TPSC_Session` structure concurrently from multiple threads. Multi-threaded TPS Clients must use platform-provided locking mechanisms to ensure that this case does not occur.
- Using the same `TPSC_MessageBuffer` structure for multiple concurrent operations.

Implementer Notes

TPS Services MUST use `TPSC_SUCCESS` to indicate success in their protocol, as this is the only way for the Implementation to determine success or failure without knowing the protocol of the TPS Service.

5 Connector Interface to Communication Stack

An implementation of the TPS Client API supports connection of backends implementing TPS Services on different types of Secure Component. To simplify the implementation of such backends, a Connector API is defined.

The Connector interface needs to support the following use cases:

- A TPS Client API Service shall be able to connect to multiple Secure Components. This implies a need to interface with multiple Communication stacks.
- Enumerate the TPS Services provided by each Secure Component.
- Perform clean-up of structures in the communications interface in the event of an unrecoverable error.

The interface has been designed assuming no more than the functionality provided by a linker of a standard C compiler.

5.1 Conceptual Architecture

Figure 5-1 outlines the conceptual architecture of TPS Client Connectors. A Connector provides an interface to a Communication stack for a particular type of Secure Component. The Communication stack may be standardized, defined by other standards bodies, or proprietary; for example:

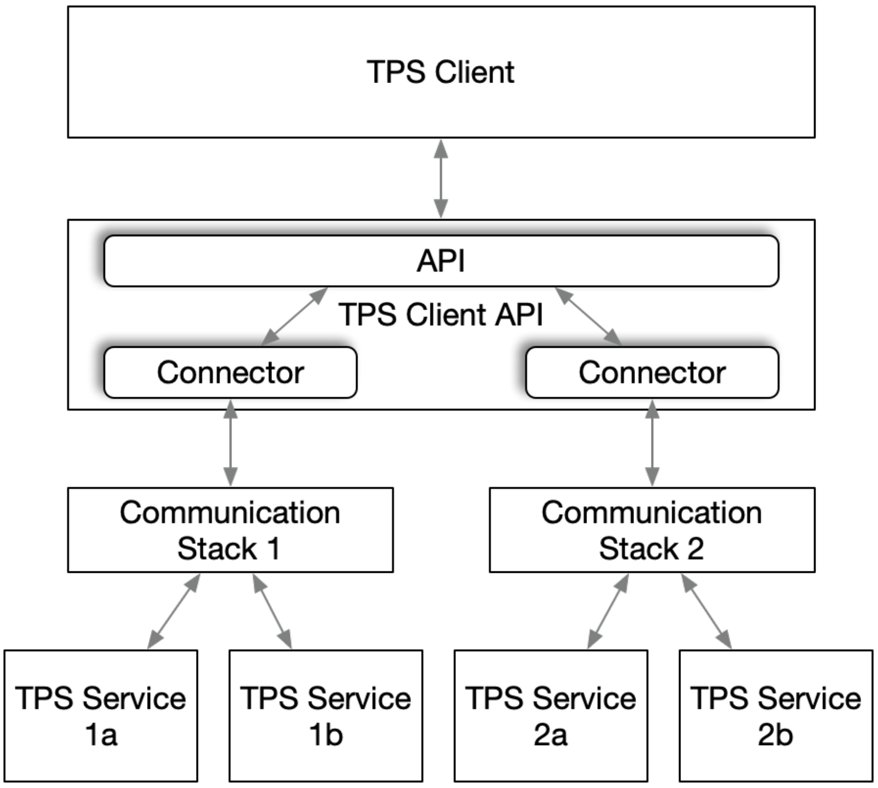
- GlobalPlatform TEE implementations use the TEE Client API ([TEE Client]).
- GlobalPlatform Secure Element implementations use the Open Mobile API ([OMAPI]).
- Trusted Platform Modules (TPMs) use the TCG Feature API ([FAPI]).
- A TEE that is not compliant with GlobalPlatform specifications may provide an alternate Communication stack.

Depending on the target device, the Communication stack may reside in the same process as the TPS Client API (e.g. is provided as a library) or it may exist within a separate process.

The Connector provides a mechanism to abstract, as far as possible, these target dependencies from both the TPS Client API itself and from the Communication stack.

The Connector is responsible for abstracting the communication mechanism between a Secure Component and the TPS Client API and for enumerating the set of services provided by a given Secure Component.

Figure 5-1: Conceptual Architecture of TPS Client Connector Interface



What is required?

- Service Name
- Session Management
- Transaction Management
- A Service must be able to enumerate any optional features that it supports. This is how we can make "configurations" work. An array of supported configurations can be returned in the `TPS_GetFeatures_Rsp` message, with each configuration fully identifying the features that the service instance can provide.

5.2 Connector Messaging

All TPS Service implementations SHALL support the following messages to assist with service discovery by client applications. The messaging defined in this section MAY be implemented in the Connector itself, in the TPS Service residing within the Secure Component, or some combination of the two.

5.2.1 TPS_GetFeatures_Req

Since: TPS Client API v1.0

A CBOR message from a client to request information about supported features.

```
TPS_GetFeatures_Req = #6.1
```

5.2.2 TPS_GetFeatures_Rsp

Since: TPS Client API v1.0

A CBOR message from a TPS Service implementation returning information about supported features.

The message MUST include information about the login methods supported for session opening.

```
TPS_GetFeatures_Rsp = #6.1 ({
      1 => svc_name,
      2 => [+ login_method],
    ? 3 => [+ profile_name ],
    $$svc_features
})

svc_name       : tstr .size 16
login_method   : uint
profile_name   : tstr

$$svc_features //= (svc_feature_label => svc_feature_type)
```

- The `svc_name` parameter is a CBOR `tstr` containing the `tps-service-name` described in section 3.2.3.
- The `login_method` parameter is a CBOR `uint` which is a value from the set of Session Login Methods listed in Table 4-3. The parameter is enclosed in an array of all of the supported Session Login Methods for a given service.
  - There SHALL be at least one `login_method` provided for any service.
  - The `uint` encoding SHOULD be canonical.
- The `profile_name` parameter is a CBOR `tstr` naming a configuration supported by a service instance. If the service instance supports at least one configuration, the enclosing array SHALL be present and SHALL contain all supported configurations.
- The `$$svc_features` parameter is defined by each service instance.

Note (Normative): The keys 0..10 and 32..127 in the `TPS_GetFeatures_Rsp` message are reserved for this specification. The keys 11..31 and 128.. can be used in the `$$svc_features` definition for each service, using the CDDL group sockets extension mechanism.

5.3 Connector API

Each Connector implementation exports a `TPSC_Connector` structure which exposes pointers to the functions provided by the Connector.

Note: The mechanism by which an implementation of the TPS Client API enumerates Connector instances from the underlying platform is out of scope of this specification.

5.4 Connector Structures

5.4.1 TPSCC_Connector

Since: TPS Client API v1.0

```
typedef struct {
    uint32_t (*connect)(uint32_t login_method,
                        const TPSC_ConnectionData *connection_data,
                        uint32_t *connection_id);
    uint32_t (*disconnect)(uint32_t connection_id);
    uint32_t (*discover_services)(uint32_t connection_id,
                                  TPSC_ServiceIdentifier *result_buf,
                                  size_t *len);
    uint32_t (*open_session)(uint32_t connection_id,
                             const TPSC_UUID *service_instance,
                             uint32_t *session_id);
    uint32_t (*close_session)(uint32_t session_id);
    uint32_t (*execute_transaction)(uint32_t session_id,
                                    uint8_t *buf,
                                    size_t buf_max_len,
                                    size_t *data_len,
                                    uint32_t *transaction_id);
    uint32_t (*cancel_transaction)(uint32_t transaction_id);
} TPSCC_Connector;
```

Description

`TPSCC_Connector` is a structure containing pointers to the functions exposed by a Connector instance.

Each Connector provides a mechanism to expose a `TPSCC_Connector` instance that provides the functions that are called by the TPS Client API when it accesses the Secure Component exposed.

5.4.1.1 cancel_transaction

Since: TPS Client API v1.0

```
TPSC_Result cancel_transaction(uint32_t transaction_id);
```

Description

The function requests the cancellation of a pending open session operation or Transaction invocation operation. As this is a synchronous API, this function must be called from a thread other than the one executing the `TPSC_OpenSession` or `TPSC_ExecuteTransaction` function.

See section 4.5.2 for additional information on cancellation semantics.

Parameters

- `transaction_id`: Identifier for the transaction that the caller wishes to cancel.

Return

- `TPSC_SUCCESS`: `transaction_id` was valid in the system and the implementation received the cancellation request.
- `TPSC_ERROR_CANCEL`: `transaction_id` was unknown but the TPS Client API implementation supports cancellation.
- `TPSC_ERROR_NOT_SUPPORTED`: The Connector implementation does not support cancellation.

Programmer Error

None

5.4.1.2 close_session

Since: TPS Client API v1.0

```
TPSC_Result close_session(uint32_t session_id);
```

Description

The function closes a session that was opened with a TPS Service.

All transactions within the session MUST have completed before calling this function.

The Implementation MUST do nothing if the `session_id` parameter is not known to the service.

The implementation of this function MUST NOT fail: After this function returns, the TPS Client must be able to consider that the session has been closed as discussed in section 3.3.3.

Parameters

- `session_id`: Identifies the session with the TPS Service.

Return

- `TPSC_SUCCESS`: Always returned in this version of the specification.

Programmer Error

None

5.4.1.3 connect

Since: TPS Client API v1.0

```
TPSC_Result connect(
  const uint32_t               login_method,
  const ConnectionData* const connection_data,
  uint32_t*                    connection_id
);
```

Description

The function opens a connection to a Secure Component through its Connector, providing login credentials if required.

Some Secure Components may not support all the available login methods, and the Connector implementation MUST return a failure value if an unsupported mechanism is requested.

If the Connector implementation requires an open connection in order to perform Service Discovery, the Secure Component MUST allow information about supported services to be provided when a caller uses `TPSC_LOGIN_PUBLIC`.

Parameters

- `login_method`: Holds one of the login methods described in section 4.4.2.
- `connection_data`: Provides additional data for those login methods that require it. See section 4.3.2.
- `connection_id`: Points to a `uint32_t` that is updated with a value that is unique to the Connector instance and can be used to identify the connection instance if required. This value is undefined in case of error and is undefined on entry.

Return

- `TPSC_SUCCESS`: Connection completed successfully. The value pointed to by `connection_id` is valid.
- `TPSC_ERROR_NOT_SUPPORTED`: The value provided for `login_method` is not supported by this Connector.
- `TPSC_ERROR_BAD_FORMAT`: The `login_method` is supported, but the Connector could not understand `connection_data`.
- `TPSC_ERROR_ACCESS_DENIED`: The combination of `login_method` and `connection_data` is supported, but the provided credentials do not allow access to the Secure Component.

Programmer Error

- `connection_id` is `NULL`.

5.4.1.4 disconnect

Since: TPS Client API v1.0

```
TPSC_Result disconnect(uint32_t connection_id);
```

Description

The function closes an open connection to a Secure Component.

Parameters

- `connection_id`: A unique identifier for a connection to the Secure Component supported by this Connector, previously returned by a call to `connect`.

Return

- `TPSC_SUCCESS`: Connection closed successfully.

Programmer Error

- The value provided for `connection_id` does not represent an open connection.

5.4.1.5 discover_services

Since: TPS Client API v1.0

```
TPSC_Result discover_services (
    const uint32              connection_id,
    TPSC_ServiceIdentifier*  result_buf,
    size_t*                   num_services
);
```

Description

This function returns the address of an array containing `TPSC_ServiceIdentifier` instances which represent the TPS Service names provided by this Connector.

Short buffer handling (see section 3.3.4) MUST be supported to cover the case where `result_buf` is not large enough to hold the returned data.

Parameters

- `connection_id`: A unique connection identifier which was obtained by a successful call to `connect`.
- `result_buf`: A pointer to a contiguous buffer of `TPSC_ServiceIdentifier` instances containing the TPS Service names provided by this Connector. This pointer MUST be valid on entry and MUST point to an allocated memory area at least `sizeof(TPSC_ServiceIdentifier) * (*num_services)`.
- `num_services`: On entry, a pointer to an integer that indicates the number of instances of `TPSC_ServiceIdentifier` that `result_buf` can hold. On successful return, points to the number of entries in the `result_buf` array.

Return

- `TPSC_SUCCESS`: The names field contains an array of `TPSC_ServiceIdentifier` instances describing valid services for this Connector and `num_services` indicates the number of services.
- `TPSC_ERROR_SHORT_BUFFER`: Provided `result_buf` was not large enough to hold the `TPSC_ServiceIdentifier` array.
- `TPSC_ERROR_SECURITY`: The caller is not authorized to access the requested service.
- `TPSC_ERROR_OUT_OF_MEMORY`: An out of memory error prevented the call from succeeding.
- `TPSC_ERROR_GENERIC`: Any other error.

Programmer Error

None

5.4.1.6 execute_transaction

Since: TPS Client API v1.0

```
TPSC_Result execute_transaction(
    const uint32_t session_id,
    const uint8_t* send_buf,
    const size_t   send_len,
    uint8_t*       recv_buf,
    size_t*        recv_len,
    uint32_t*      transaction_id
);
```

Description

C callable API to request a Service to perform a transaction with the provided parameters.

Parameters

- `session_id`: Identifies the session requesting the service. Since a session is bound to a service identifier, this identifies the target service for the operation.
- `send_buf`: Must point to a readable memory area of at least length `send_len` bytes. It contains the message being sent to the service.
- `recv_buf`: Must point to a writable memory area of at least length `recv_len` bytes. It will contain the response from the service on return. `recv_len` is updated with the length of the returned data. Short buffer handling (see section 3.3.4) MUST be supported to cover the case where `recv_buf` is not large enough to hold the returned data.
- `transaction_id`: Must be writable. The value on entry and in the case of failure is undefined. On successful return it contains a transaction identifier which can be used to cancel the transaction.

Return

- `TPSC_SUCCESS`: Session was successfully opened.
- `TPSC_ERROR_NO_DATA`: `send_buf`, `recv_buf`, or `session` is `NULL`, `send_len` is zero.
- `TPSC_ERROR_BAD_FORMAT`: `send_buf` or `recv_buf` was not initialized before the function was called.
- `TPSC_ERROR_SHORT_BUFFER`: The buffer allocated in `recv_buf` is not large enough to contain the response. In this case, the handling in section 3.3.4 applies and `recv_buf->size` contains the size of the buffer required to hold the TPS Service response message.

Programmer Error

The following usage of the API is a programmer error:

- Calling with a `session_id` that is not an open session.
- Using the same `session_id` concurrently for multiple operations.
- Calling with invalid content in the `message` field of the `send_buf` structure.
- Using the same `send_buf` or `recv_buf` structure concurrently for multiple operations.

5.4.1.7 open_session

Since: TPS Client API v1.0

```
TPSC_Result open_session(
  const uint32_t  connection_id,
  const UUID*     service_instance,
  uint32_t*       session_id
);
```

Description

The function creates a new session with a specific TPS Service instance. This session is identified using the value returned in the `session_id` parameter, which is guaranteed to be unique for the Secure Component which hosts the service.

Parameters

- `connection_id`: A unique connection identifier which was obtained by a previous successful call to `connect`.
- `service_instance`: A UUID which identifies a unique TPS Service on the Secure Component that is accessed via the Connector instance. It is a value that was previously returned in the `service_identifier` field of a `TPSC_ServiceIdentifier`.
- `session_id`: Holds a session identifier that uniquely identifies the session creates with the TPS service. On entry or on failed return, the value is undefined. On successful return, it holds a session identifier that is used in transactions between the client and the service.

Return

- `TPSC_SUCCESS`: Session was successfully opened.
- `TPSC_ERROR_BAD_IDENTIFIER`: The value provided for `service_instance` does not identify a `tps-service-instance` on the secure component associated with this Connector.
- `TPSC_ERROR_NULL_POINTER`: `service_instance or session_id was NULL`.

Programmer Error

None

Annex A [Informative] Rust Language API

This annex defines an optional Rust language version of the TPS Client API. It may be useful where the client application is implemented in Rust, from both a performance and correctness perspective.

Rust APIs are organized as Crates, which can contain Modules. For each API element, we specify the Crate and module in which it is defined.

Note: A future version of this specification is expected to define a normative Rust language API specification.

A.1 Behavior

Exported Rust functions and data types have identical externally visible behavior to their C counterparts, with the exception that Rust functions use an idiomatic error handling mechanism that is functionally equivalent to that provided by the C API.

A.2 Mapping C API Names to Rust Names

Rust places strong requirements on the naming conventions for program elements such as functions and structures, and provides a namespace mechanism that eliminates namespace clashes. As such, names in the Rust APIs differ from those in the C language API described previously. C names can be mapped to Rust names as follows:

- Function names are all lowercase with underscores between words, with no TPSC prefix.
  - e.g. `TPSC_ExecuteTransaction` becomes `execute_transaction` in the Rust API.
- Structure and constant names are prefixed with TPSC prefix in C. No such prefix is used in Rust.
  - e.g. `TPSC_ServiceIdentifier` becomes `ServiceIdentifier` in Rust.

A.3 Rust Data Types

The exported Rust data types are found in the `c_structs` module of the enclosing Crate as they are shared between the C and Rust APIs.

A.3.1 mod c_structs

Since: TPS Client API v1.0

```
pub mod c_structs {
    #[repr(C)]
    pub enum ConnectionData {
        None,
        GID(u32),
        Proprietary(*const c_void)
    }

    #[repr(C)]
    pub enum ServiceBound {
        Inclusive(ServiceVersion),
        Exclusive(ServiceVersion),
        NoBound
    }

    #[repr(C)]
    pub struct ServiceIdentifier {
        pub service_instance: UUID,
        pub service_id: UUID,
        pub secure_component_type: UUID,
        pub secure_component_instance: UUID,
        pub service_version: ServiceVersion,
    }

    #[repr(C)]
    pub struct ServiceRange {
        pub lowest_acceptable_version: ServiceBound,
        pub first_excluded_version: ServiceBound,
        pub last_excluded_version: ServiceBound,
        pub highest_acceptable_version: ServiceBound,
    }

    #[repr(C)]
    pub struct ServiceSelector {
        pub service_id: UUID,
        pub secure_component_type: UUID,
        pub secure_component_instance: UUID,
        pub service_version_range: ServiceRange,
    }

    #[repr(C)]
    pub struct ServiceVersion {
        pub major_version: u32,
        pub minor_version: u32,
        pub patch_version: u32,
    }

    #[repr(C)]
    pub struct Session {
        pub service_id: *const UUID,
        pub session_id: u32,
        pub imp: SessionPriv,
    }

    #[repr(C)]
    pub struct MessageBuffer {
        pub message: *mut u8,
        pub size: usize,
        pub maxsize: usize,

        pub imp: MessageBufferPriv,
    }

    #[repr(C)]
    pub struct UUID {
        pub bytes: [u8; 16]
    }
}
```

A.3.2 Additional Structures

A.3.2.1 mod r_structs

Since: TPS Client API v1.0

The `r_structs` module defines a structure, `UnsafeMessageBuf`, which can be straightforwardly constructed from a `MessageBuffer`, but which has more straightforward Rust ergonomics. `UnsafeMessageBuffer` is not C language FFI compatible.

Note: In the code below, lifetime annotations have been removed for simplicity. Real code will require annotation for the buffer lifetime and may require additional lifetime annotation.

```
pub mod r_structs {
    pub struct UnsafeMessageBuf {
        msg_len: usize,
        buffer: &[u8],
        imp: MessageBufferPriv
    }

    impl From for UnsafeMessageBuf {
        fn from(mb: MessageBuffer) -> Self {
            UnsafeMessageBuf {
                buffer = unsafe {from_raw_parts_mut(mb.message, mb.maxsize)},
                msg_len: mb.size,
                imp: mb.imp
            }
        }
    }
}
```

As the name implies, `UnsafeMessageBuf` is not safe for use in multi-threaded Rust code, and it is typically wrapped using mechanisms to ensure thread-safety and safe inner mutability (`RefCell`, `Arc`, `Mutex`, or similar, depending on the use-case).

The safe, wrapped variant of `UnsafeMessageBuf` is the `MessageBuf` type which is used in the Rust language API definitions in section A.6. In this version of the specification, `MessageBuf` is implementation-defined, but it is expected to behave as though it implements the following trait:

```
pub trait SafeMessageBuf {
    type OwnerGuard;

    fn new(buf: &mut [u8]) -> Self;
    fn new_from_unsafe_message_buf(u_buf: UnsafeMessageBuf) -> Self;
    unsafe fn new_from_mut_ptr(buf: *mut u8, len: usize) -> Self;
    fn lock(&self) -> Self::OwnerGuard;
    fn set_len(&self, l: usize) -> Result<(), TPSError>;
}
```

In addition, it is expected to support mutable and immutable `Iterator` traits.

A.4 Constants

The exported Rust constants are split across three modules of the `tps_client_common` Crate as they are shared between the C and Rust APIs.

A.4.1 mod c_errors

Since: TPS Client API v1.0

```
pub mod c_errors {
    pub const SUCCESS: u32 = 0x00000000;
    pub const ERROR_GENERIC: u32 = 0xF0090000;
    pub const ERROR_ACCESS_DENIED: u32 = 0xF0090001;
    pub const ERROR_CANCEL: u32 = 0xF0090002;
    pub const ERROR_BAD_FORMAT: u32 = 0xF0090003;
    pub const ERROR_NOT_IMPLEMENTED: u32 = 0xF0000004;
    pub const ERROR_NOT_SUPPORTED: u32 = 0xF0090005;
    pub const ERROR_NO_DATA: u32 = 0xF0090006;
    pub const ERROR_OUT_OF_MEMORY: u32 = 0xF0090007;
    pub const ERROR_BUSY: u32 = 0xF0090008;
    pub const ERROR_COMMUNICATION: u32 = 0xF0090009;
    pub const ERROR_SECURITY: u32 = 0xF009000A;
    pub const ERROR_SHORT_BUFFER: u32 = 0xF009000B;
    pub const ERROR_DEPRECATED: u32 = 0xF009000C;
    pub const ERROR_BAD_IDENTIFIER: u32 = 0xF009000D;
    pub const ERROR_NULL_POINTER: u32 = 0xF009000E;
    pub const ERROR_BAD_STATE: u32 = 0xF009000F;
    pub const ERROR_TIMEOUT: u32 = 0xF0090010;
    pub const ERROR_PLATFORM: u32 = 0xF0090011;
    pub const ERROR_RUNTIME_ERROR: u32 = 0xF0090012;
}
```

A.4.2 mod c_login

Since: TPS Client API v1.0

```
pub mod c_login {
    pub const LOGIN_PUBLIC: u32 = 0x00000000;
    pub const LOGIN_USER: u32 = 0x00000001;
    pub const LOGIN_GROUP: u32 = 0x00000002;
    pub const LOGIN_APPLICATION: u32 = 0x00000004;
    pub const LOGIN_USER_APPLICATION: u32 = 0x00000005;
    pub const LOGIN_GROUP_APPLICATION: u32 = 0x00000006;
    pub const CONNECTIONDATA_NONE: u32 = 0x00000000;
    pub const CONNECTIONDATA_GID: u32 = 0x00000001;
    pub const CONNECTIONDATA_LAST_ITEM: u32 = 0x7fffffff;
}
```

A.4.3 mod c_uuid

Since: TPS Client API v1.0

```
pub mod c_uuid {
    pub const UUID_NIL: UUID = UUID {
        bytes: [0; 16]
    };
    pub const UUID_NAMESPACE: UUID = UUID {
        bytes: [0x99, 0x13, 0x67, 0x3c, 0x23, 0x32, 0x42, 0x2c,
            0x82, 0x13, 0x1e, 0xc1, 0xf7, 0x49, 0x36, 0xe8]
    };
    pub const UUID_SC_TYPE_GPD_TEE: UUID = UUID {
        bytes: [0x59, 0x84, 0x68, 0x75, 0x1e, 0x02, 0x53, 0xc8,
            0x92, 0x2f, 0x5d, 0x60, 0xdd, 0x10, 0x3a, 0x58]
    };
    pub const UUID_SC_TYPE_GPC_SE: UUID = UUID {
        bytes: [0xbd, 0xd6, 0x58, 0xfa, 0x44, 0xc1, 0x5e, 0x59,
            0xb3, 0xa1, 0x1a, 0x8f, 0x03, 0x8c, 0xeb, 0x50]
    };
    pub const UUID_SC_TYPE_GPP_REE: UUID = UUID {
        bytes: [0xd2, 0xdc, 0x12, 0x0c, 0x3e, 0x4a, 0x5b, 0x1f,
            0xbe, 0xce, 0xdf, 0x38, 0x25, 0xc9, 0x33, 0xae]
    };
}
```

A.5 Errors

Since: TPS Client API v1.0

As discussed previously, Rust functions are provided with an idiomatic mechanism for handling errors, along with a mechanism to transform Rust errors into the values expected by the C API.

The error handling mechanism is implemented in the `error` module of the `tps_client_api` Crate.

```
pub enum TPSError {
    GenericError,
    AccessDenied,
    Cancel,
    BadFormat,
    NotImplemented,
    NotSupported,
    NoData,
    OutOfMemory,
    Busy,
    CommunicationError,
    SecurityError,
    ShortBuffer(usize),
    Deprecated,
    BadIdentifier,
    NullPointer,
    BadState,
    Timeout,
    Platform,
    RuntimeError
}
```

The `Into` Trait has the following instance for `TPSError`:

- `impl Into<u32> for TPSError`.

A.6 Functions

Since: TPS Client API v1.0

The TPS Client API functions are implemented in the `connector` module of the `tps_client_api` Crate.

```
pub fn cancel_transaction(_transaction: &MessageBuf)
-> Result<(), TPSError>

pub fn close_session(_session: &Session)
-> Result<(), TPSError>

pub fn discover_services(
    _service_selector: &ServiceSelector,
    _service_array: &mut [ServiceIdentifier],
) -> Result<usize, TPSError>

pub fn execute_transaction(
    _session: &Session,
    _send_buffer: &MessageBuf,
    _recv_buffer: &MessageBuf,
) -> Result<(), TPSError>

pub fn open_session(
    _service_uuid: &UUID,
    _connection_data: Option<&ConnectionData>,
) -> Result<Session, TPSError>
```

Annex B [Informative] Sample Code for Calling the TPS API from a Client Application

```
#include <stdio.h>
#include <stdint.h>
#include "tpsc_client_api.h"

// Defines a ROT13 service called "GPP ROT13" using the normative namespace
// 87bae713-b08f-5e28-b9ee-4aa6e202440e
#define SERVICE_ID_GPP_ROT13 { .bytes = { 0x87, 0xba, 0xe7, 0x13, 0xb0, 0x8f, 0x5e, 0x28, \
                                          0xb9, 0xee, 0x4a, 0xa6, 0xe2, 0x02, 0x44, 0x0e } }

#define TRANSACTION_BUFFER_SIZE (256)
#define ARRAY_SIZE(val, type)   (sizeof(val)/sizeof(type))

/* A real program would use a CBOR encoder and decoder. For simplicity the CBOR for input to the
 * Service and the expected output is hard-coded.
 *
 * The input (in CBOR Diagnostic format) is: 10({1:"Thisgoestoeleven"}).
 * Expected output (in CBOR diagnostic format): 10({1:"Guvftbrfgbryrira"})
 */
#define INPUT_MSG  {0xCA, /* tag(10) */\
                    0xA1, /* map(1) */\
                    0x01, /* unsigned 1 */\
                    0x70, /* tstr(16) */\
                    0x54, 0x68, 0x69, 0x73, 0x67, 0x6F, 0x65, 0x73, 0x74, \
                    0x6F, 0x65, 0x6C, 0x65, 0x76, 0x65, 0x6E /* "Thisgoestoeleven" */ \
                    }
#define EXPECT_MSG {0xCA, /* tag(10) */\
                    0xA1, /* map(1) */\
                    0x01, /* unsigned 1 */\
                    0x70, /* tstr(16) */\
                    0x47, 0x75, 0x76, 0x66, 0x74, 0x62, 0x72, 0x66, 0x67, \
                    0x62, 0x72, 0x79, 0x72, 0x69, 0x72, 0x61 /* "Guvftbrfgbryrira" */ \
                    }


uint32_t DoServiceDiscovery(TPSC_ServiceIdentifier* service_container) {

    TPSC_ServiceSelector selector = {
            .service_id = SERVICE_ID_GPP_ROT13,
            .secure_component_instance = TPSC_UUID_NIL,
            .secure_component_type = TPSC_UUID_NIL,
            .service_version_range = {
                    .lowest_acceptable_version = { .tag = Inclusive,
                                                   .inclusive = {
                                                        .major_version = 0,
                                                        .minor_version = 0,
                                                        .patch_version = 1
                    }},
                    .first_excluded_version = { .tag = Inclusive,
                                                .inclusive = {
                                                    .major_version = 1,
                                                    .minor_version = 1,
                                                    .patch_version = 1
                    }},
                    .last_excluded_version = { .tag = Exclusive,
                                               .exclusive = {
                                                    .major_version = 1,
                                                    .minor_version = 2,
                                                    .patch_version = 0
                    }},
                    .highest_acceptable_version = { .tag = Exclusive,
                                                    .exclusive = {
                                                        .major_version = 2,
                                                        .minor_version = 0,
                                                        .patch_version = 0
                    }}
            }
    };
    size_t num_services;
    static TPSC_ServiceIdentifier array[3];
    size_t num_services = sizeof(services_available) / sizeof(TPSC_ServiceIdentifier);

    uint32_t retval = TPSC_DiscoverServices(&selector, &num_services, &service_array);
    service_container = &array[0];
    return retval;
}

int main(int argc, char** argv) {
    TPSC_ServiceIdentifier svc_id;

    if (DoServiceDiscovery(&svc_id) == TPSC_SUCCESS) {

    TPSC_Session session;
    if (TPSC_OpenSession(&(svc_id.service_instance), TPSC_LOGIN_PUBLIC, NULL, &session) ==
TPSC_SUCCESS) {
            void *send_buffer = malloc(TRANSACTION_BUFFER_SIZE);
            void *recv_buffer = malloc(TRANSACTION_BUFFER_SIZE);
            TPSC_MessageBuffer send_buf;
            TPSC_MessageBuffer recv_buf;
            if ((TPSC_InitializeTransaction(&send_buf, send_buffer,
                    TRANSACTION_BUFFER_SIZE) == TPSC_SUCCESS) &&
            (TPSC_InitializeTransaction(&recv_buf, recv_buffer,
                    TRANSACTION_BUFFER_SIZE) == TPSC_SUCCESS)){
            PrepareMessage(send_msg, 20 /*ARRAY_SIZE(send_msg, uint8_t)*/, send_buffer,
                    TRANSACTION_BUFFER_SIZE);
            send_buf.size = 20; //sizeof(ARRAY_SIZE(send_msg, uint8_t));
            if (TPSC_ExecuteTransaction(&session, &send_buf, &recv_buf) == TPSC_SUCCESS) {
                    PrintMessage("Received Message", recv_buf.message, recv_buf.size);
            } else {
                    printf("Transaction failed!");
            }
            }
    }
    } else {
    printf("Service discovery failed");
    }
}
```