



CLOUD
SIGNATURE
CONSORTIUM

Standard

Architectures, Protocols and API Specifications for Remote Signature applications

Public pre-release version 0.1.7.9 rev. PR (2017-02)

All rights reserved – Copyright 2016-2017 **Cloud Signature Consortium**

Promoters: Adobe Systems, Asseco Data Systems, Certinomis, Cryptolog, D-Trust, Graz University of Technology, InfoCert, Intarsys Consulting, Intesi Group, Izenpe, SafeLayer Secure Communications, SwissSign, Unibridge.



Contents

Foreword	4
Revision history	4
Acknowledgements	4
Introduction.....	5
Legal notices	5
1 Scope	6
2 Requirements Language	7
3 References	7
3.1 Normative references.....	7
3.2 Informative references	7
4 Terms and definitions.....	8
5 Conventions.....	8
6 Architectures and use cases	9
6.1 Supported architectures.....	9
7 Entities and components of a Remote Signature solution.....	10
8 Introduction to the Remote Service Protocols API.....	11
8.1 Format and syntax of the API	11
8.2 Remote Service Base URI.....	11
8.3 Integrity and confidentiality	11
8.4 Remote Service Information.....	12
9 Authentication and authorization	12
9.1 Service authorization and authentication.....	12
9.2 Credential authorization.....	13
9.3 OAuth 2.0 Authorization.....	14
OAuth 2.0 Authorization Code [oauth2/authorize].....	16
OAuth 2.0 Implicit Grant [oauth2/authorize]	19
10 Creating a Remote Signature.....	22
10.1 Multi-signature Transactions.....	22
11 Error handling.....	22
11.1 Error messages	23
12 The Remote Service APIs	24
info	25
auth/login	27
auth/revoke	29



oauth2/token	31
credentials/list	34
credentials/info	36
credentials/authorize	41
credentials/extendTransaction	43
credentials/sendOTP	44
signatures/signHash	45
signatures/timestamp	47
13 Interaction among elements and components	49
13.1 Acquire the context of a RSSP	49
13.2 RSSP service authorization using a username and password	49
13.3 RSSP service authorization using OAuth2 with Implicit Grant flow	50
13.4 RSSP service authorization using OAuth2 with Authorization Code flow	50
13.5 RSSP service authorization using OAuth2 with Client Credentials flow	50
13.6 Get the list of credentials available for a group or community of users	51
13.7 Create a remote signature with a credential protected by an implicit authorization	51
13.8 Create a remote signature with a credential protected by a PIN	52
13.9 Create a remote signature with a credential protected by an “offline” OTP	53
13.10 Create a remote signature with a credential protected by an “online” OTP	54
13.11 Create a remote signature with a credential protected by PIN and OTP	55
13.12 Create a remote signature with a credential protected by a biometric factor	56
13.13 Create multiple remote signatures from a list of hash values	57
13.14 Create a remote multi-signatures transaction	58
13.15 Create a remote signature with long-term validity profile	59



Foreword

This document is a work by members of the Cloud Signature Consortium, a collaborative initiative among industry and academic organizations for building upon existing knowledge of solutions, architectures and protocols for Remote Electronic Signatures, also defined as Cloud-based Digital Signatures.

The Cloud Signature Consortium has developed the present specification to make these solutions interoperable and suitable for uniform adoption in the global market, in particular – but not exclusively – to meet the requirements of the European Union's Regulation 910/2014 on Electronic Identification and Trust Services (eIDAS), which formally took effect on 1 July 2016.

Revision history

Version	Date	Version change details
0.1.7.9-PR	14/02/2017	Public pre-release for comment

Acknowledgements

This work is the result of the contributions of several individuals from the promoting members of the Cloud Signature Consortium. In particular, the following people have provided a significant contribution to the drawing up and revision of the present document:

Giuseppe Damiano, Andrea Valle, Luigi Rizzo, Franck Leroy, Andrea Röck, Thomas Pielczyk, Michael Traut, Jon Ølnes, Peter Lipp, Patrycja Wiktorczyk, Ała Stoliarowa-Myć, Marcin Szulga, Arno Fiedler, Kapil Khattar, Meena Muralidharan, Marc Kaufman, Iñigo Barreira, Bernd Wild, Dr. Kim Nguyen, Prof. Reinhard Posch, Mangesh Bhandarkar, Cornelia Enke, Klaus-Dieter Wirth, Carlos Ares, Giuliana Marzola, Arkadiusz Lawniczak, Enrico Entschew, Andreas Vollmert, Luca Boldrin, Håvard Grindheim.



Introduction

For a long time, transactional e-services have been designed for typical end-user devices such as desktop computers and laptops. Accordingly, existing digital signature solutions are tailored to the characteristics of these devices as well. This applies to smart card and USB token-based solutions. These traditional signature solutions implicitly assume that the user accesses e-services from a desktop or laptop computer and in addition uses a smart card or token to create any required digital signatures. Recently, this assumption is not valid any longer. During the past few years, smartphones, tablets and other mobile end-user devices have started to replace desktop and laptops computers.

This situation raises several challenges for e-services: smart cards and tokens cannot be easily connected to smartphones and other mobile devices, or cannot at all. For instance, smartphones usually do not provide support for USB devices, which is the typical technology for smart card based solutions.

In this regard, recent regulations in various Regions worldwide – like eIDAS in the European Union – have introduced the concept of electronic signatures that are created using a “remote signature creation device”, which means that the signature device is not anymore a personal device under the physical control of the user, but rather it is replaced by cloud-based services offered and managed by a trusted service provider.

This is in summary the context of the Cloud Signature Consortium, aimed at defining a common architecture, building blocks and communication protocols aimed at creating a standard API to integrate the essential components of a Remote Signature solution established among different service providers and consumers.

Where the context of the eIDAS Regulation is applicable, this specification, and the term “Remote Signature solution” herein developed, aim to cover solutions for Remote Electronic Signatures and Remote Electronic Seals, in the domains of both Qualified and Advanced Electronic Signatures.

Legal notices

The Consortium seeks to promote and encourage broad and open industry adoption of the specifications, including making available licenses for use under Fair, Reasonable and Non-Discriminatory (FRAND) terms to non-members of the consortium for adoption in any transactions between individuals, companies, governmental bodies and other organizations.

The present document does not create legal rights and does not imply that intellectual property rights are transferred to the recipient or other third parties. The adoption of the specification contained herein does not constitute any rights of affiliation or membership to the Cloud Signature Consortium.

This document is provided “as is” and the Cloud Signature Consortium and its members are not responsible for any errors or omissions. Reuse and repurpose of content from the present document are subject to written authorization from the Cloud Signature Consortium.

The use of the Cloud Signature Consortium Logo is reserved to members of the Cloud Signature Consortium.

Questions and comments to this document can be sent to ask@cloudsignatureconsortium.org.

1 Scope

When digital signatures are created within a device, the interfaces and functions are standardized, e.g. the API used by the application program to access the signature creation libraries and the interface to the smart card or similar device (if a device is used) holding the signing key. When digital signatures move to the cloud, the functions needed to create a digital signature can be distributed across several service instances, each carrying out one or more steps in the signature creation process. The interfaces between such services are however not standardized.

The Cloud Signature Consortium aims to fill this gap in standardization by defining the architectural design, communication protocols, application programming interfaces, data structures, and technical requirements needed to establish interoperable solutions for cloud-based digital signatures. While these specifications are applicable in a wide variety of use cases with different security requirements, the fulfilment of requirements imposed by the eIDAS Regulation of the EU are particularly addressed, supporting the creation of “advanced” or “qualified” electronic signatures and electronic seals in the cloud.

This document contains technical specifications that are intended for use by implementers of services for digital signatures in the cloud and by a variety of applications consuming these services. By implementing their services according to these specifications, service providers can ensure that services are applicable as parts of complete digital signature systems in the cloud in a plug and play manner.

The specifications of the Cloud Signing Consortium will be implemented and tested by its members. When sufficient experience is gained, the consortium aims to submit the specifications for formal standardization by a European or international standards organization. Existing standards and open specifications are taken into account by the consortium as far as these are applicable.

The Consortium seeks to promote and encourage broad and open industry adoption of the specifications, including making available licenses for use under Fair, Reasonable and Non-Discriminatory (FRAND) terms to non-members of the consortium for adoption in any transactions between individuals, companies, governmental bodies and other organizations.

The following are out of scope of this specification:

- Policy requirements for (qualified and other) service providers; this is an area of standardization covered by ETSI.
- Signature and certificate formats; use of the standards specified by ETSI is recommended.
- Signature validation; this will be addressed in future specifications from the Consortium.
- Security evaluation and requirements for hardware components used to hold signing keys (HSM – hardware security module); this is being standardized by CEN in Europe and FIPS in the USA.
- Internal functionality and internal interfaces in one service provider’s systems.

Note that the current specifications only cover architectures where the signing key is held “in the cloud”, i.e. by a signature creation device managed by a service provider. Architectures where the signing key is in the hand of the signer, stored in the user’s device or in an attached smart card or similar, are not covered as a particular case. The consortium will consider the need for further specifications covering situations where a user device holding the signing key interacts with cloud services for digital signature creation, e.g. cloud services may be used for document storage, hash computation, and signature formatting.

2 Requirements Language

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

3 References

3.1 Normative references

The following documents, in whole or in part, are normatively referenced in this specification and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

< List of Normative references to be completed >

RFC 2119

RFC 2253

RFC 2459

RFC 2617

RFC 3066

RFC 3161

RFC 4627

RFC 5246

RFC 5816

RFC 6749

RFC 6750

ISO 3166-1

3.2 Informative references

The following documents, in whole or in part, are informatively referenced in this specification and may be a useful contribution for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

< List of Informative references to be completed >

RFC 3447

eiDAS Regulation n. 910/2014

CEN EN 419 241-1

ETSI SR 019 020



4 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

authentication factor: data that represents some secure information used to identify or authenticate a user, for example a password or PIN.

credential: cryptographic object and related data used to support remote digital signatures over the Internet. Consists of the combination of a public/private key pair (also named “signing key” in CEN EN 419 241-1) and a X.509 public key certificate managed by a remote signing service provider on behalf of a user.

HSM: Hardware Security Module

Remote Service: service implementing the API described in this specification and delivered on the Internet.

Remote Signing Service Provider (RSSP): service provider managing a set of credentials on behalf of multiple users and allowing them to create a remote signature with a stored credential.

NOTE: A remote signing service provider typically operates an HSM (or functionally equivalent multi-user secure device) and an authentication service. It manages the users and provides a signing service that can be accessed over the Internet by means of the API described in this specification.

RSCD: Remote Signature Creation Device

SAD: Signature Activation Data

secure element: equivalent to authentication factor.

signature application: client application or service calling the RSSP to create a remote signature.

signature application provider: service provider managing a signature application and offering it as a service over the Internet or other communication channel.

signature creation service provider (SCSP): equivalent to remote signing service provider (RSSP).

signing identity: equivalent to credential.

5 Conventions

This document uses the following text conventions to help identify various types of information.

Text convention	Example
The pipe character () indicates a possible value for selection or outcome and shall be interpreted as “or”.	YES NO
Text shown with <code>Courier font</code> is example code.	POST /credentials/info HTTP/1.1
Text shown with bold within text paragraphs indicate the name of an API method.	credentials/list
Text shown with <i>italic</i> within text paragraphs indicate the name of an API input or output parameter.	<i>access_token</i>

6 Architectures and use cases

The present document and the protocols defined herein aim to support different use cases. However, they focus on the scenario of remote signing defined for example as “the creation of remote electronic signatures, where the electronic signature creation environment is managed by a trust service provider on behalf of the signatory” [EU Regulation 910/2014, whereas §52].

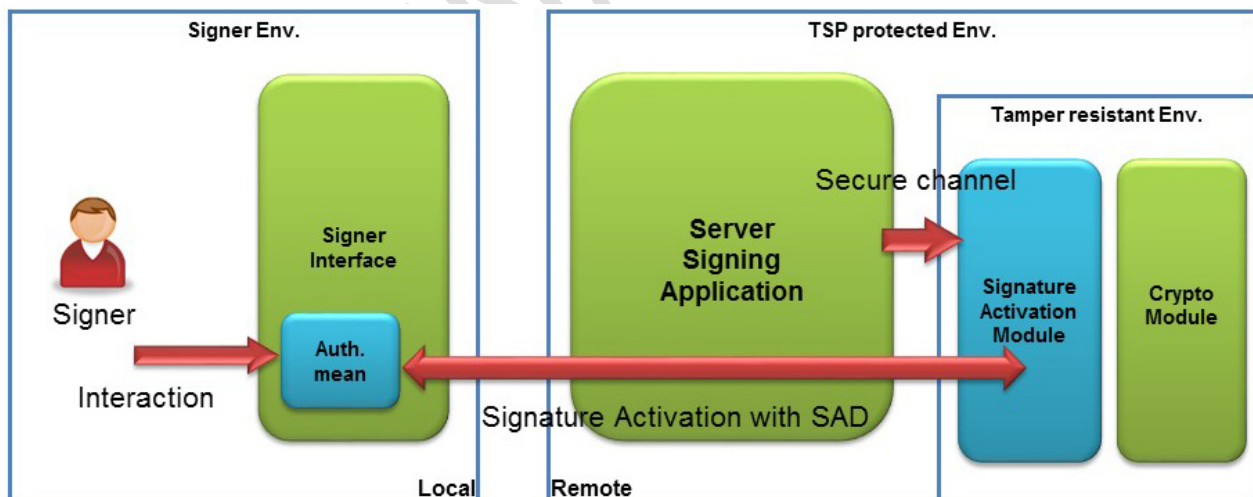
This means that other scenarios for signing in distributed environments assisted by remote servers – like those described in ETSI SR 019 020 (“Standards for AdES digital signatures in mobile and distributed environment”) – are not covered in the present version of this specification. In particular, use cases where the signing key is contained within a signer's personal device are not covered: for example, signing a document located on a server with a private key contained in a mobile SIM card, or in a cryptographic device connected to a personal computer. These are relevant use cases, although not fitting in the core definition of “remote signature”, so they may be specifically covered in future updates of the specification.

6.1 Supported architectures

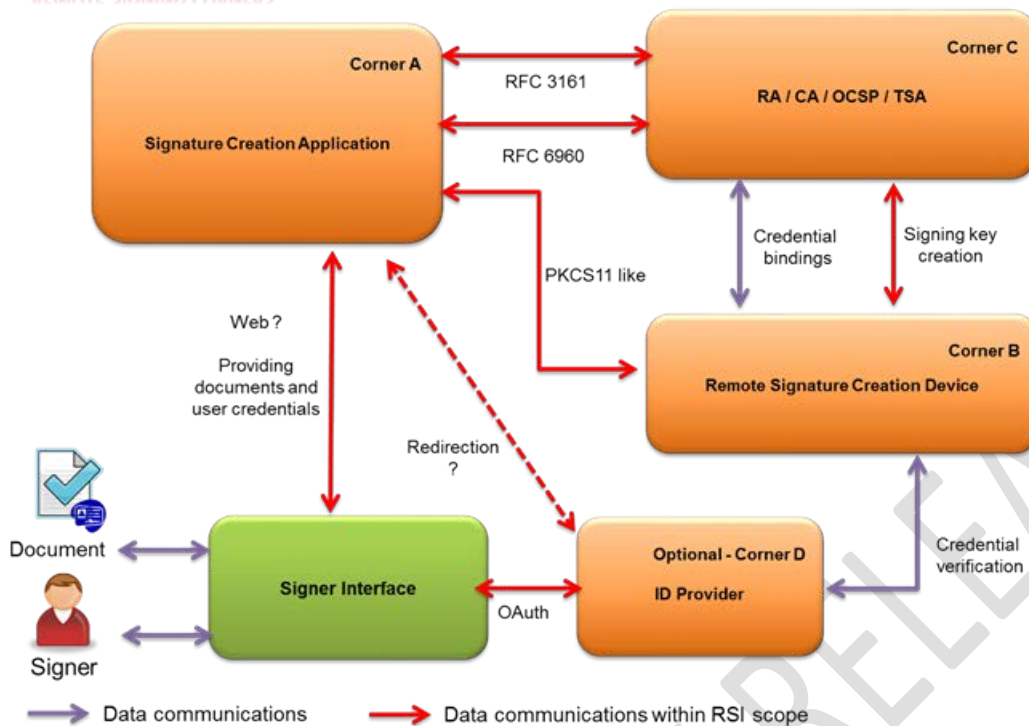
The present document focuses on the following three types of architecture:

- A signature application that is connected to a Remote Signing Service Provider hosting the Remote Signature Creation Device and to another Trust Service Provider providing CA/RA/TSA services;
- A signature application that is connected to a Remote Signing Service Provider hosting the Remote Signature Creation Device and also providing CA/RA/TSA services;
- A signature application provider that is hosting the Remote Signature Creation Device and that is connected to a Trust Service Provider providing CA/RA/TSA services.

<The following diagrams will be updated>



REMOTE SIGNING CORNERS



NOTE: In case the signature application provider hosts the Remote Signature Creation Device and also provides CA/RA/TSA services, then all the protocols described herein would only apply to communications among “internal” building blocks of the provider, and would not be exposed to additional providers or consumers. This may reduce the need for interoperability that this specification aims at achieving, but such provider could still obtain a benefit from implementing these APIs.

7 Entities and components of a Remote Signature solution

The architectures for Remote Signature solutions considered in this specification are based on elements, functional components and building blocks defined as follows:

<List to be reviewed and completed with additional descriptions>

CAS: Credential Authorization Service

RSCD: Remote Signature Creation Device

RSS: Remote Signing Service

SA: Signature Application

SAM: Signature Activation Module

SCA: Signature Creation Application

SSA: Server Signing Application



8 Introduction to the Remote Service Protocols API

An Application Programming Interface (API) is the way Web applications and services talk to each other. Technically speaking, an API is a set of programming instructions for accessing a Web-based software application or service. This specification is intended for use as an interface by software components to communicate with one another. It is important to note that an API is a software-to-software programming interface; APIs enable applications and services to talk to each other without any user intervention.

The Remote Signature Protocols API allows a signature application to communicate with a remote service via the Internet by leveraging a sequence of calls to methods. The API is a programming interface, defining how two entities communicate, and the back and forth calls constitute the communication protocols between the applications. The design goal is to provide a simple and consistent API.

8.1 Format and syntax of the API

This specification defines Web services APIs that are based on technical standards and protocols such as HTTP and JSON. This API uses HTTP POST requests with JSON payload and JSON responses. JSON is an open-standard media type format as defined by RFC 4627 that uses human-readable text to transmit data objects consisting of attribute-value pairs. These properties make JSON an ideal data-interchange language which is used as the most common data format for asynchronous communications.

The functions offered by the Remote Service are represented by HTTP RPC endpoints accepting arguments as JSON in the request body and returning results as JSON in the response body. For this reason, the HTTP header of the invocation method shall include a `Content-Type: application/json` header.

8.2 Remote Service Base URI

The Remote Service Base URI defines the style and format of the HTTP endpoint URI of a Remote Service conforming to this specification.

The Base URI contains the version number of the APIs that is implemented by the Remote Service Provider. In the case of this preliminary specification, the version shall be “v0”. Future versions of this specification may maintain retro compatibility with previous versions, but may also override them.

`https://service.domain.org/csc/v0/`

The `service.domain.org` address is used as an example here and should be replaced by the URL registered by the Remote Service Provider. The URI fragments of the API methods documented in this specification shall be concatenated to the Base URI. An exception is given by the OAuth 2.0 authorization methods, which can use a URL different from the Base URI.

8.3 Integrity and confidentiality

A Remote Service conforming to this specification shall always implement Transport Layer Security (TLS) in order to ensure the integrity and confidentiality of the communications. This prevents easy eavesdropping or impersonation if authentication credentials are hijacked. Another advantage of always using TLS is that guaranteed encrypted communications simplifies the authentication schemes, so for example simple mechanisms like Basic HTTP authentication can be used because the security elements (username and password) are always transmitted over an encrypted channel.

The only exception to using TLS communications is when the communication channel between the signature application and the Remote Service already implements other methods to guarantee integrity and confidentiality, for example using a VPN connection. In that case, TLS may not be implemented.



TLS 1.2 as described in RFC 5246 is, at the time of this writing, the latest version of TLS. TLS 1.2 shall be implemented by Remote Services conforming to this specification and is the recommended mechanism to use as it provides access to advanced cipher suites that support elliptic curve cryptography and AEAD block cipher modes. The previous versions 1.1 of TLS may be used, but, although providing a broader interoperability, it is also increasingly less secure. TLS 1.0 is considerably less secure and some security certifications like PCI DSS 3.1 explicitly forbid it, so Remote Services should not support it.

All versions of SSL, the security protocol in use before TLS, are considered insecure. Remote Services conforming to this specification shall not implement SSL.

8.4 Remote Service Information

This specification defines a standard and interoperable protocol to connect a client application to a Remote Service. Other similar specifications exist in the industry, but they are typically proprietary and incompatible, so if a Signature Application wants to support multiple Remote Services, then the development effort would increment significantly.

This specification has been designed to support modular services that may be implemented in line with the capacity and mission of the provider. This means that a Remote Service that supports this specification may implement only a particular subset of the API methods defined herein. In order to support this approach, this specification defines the **info** method, which all Remote Services shall implement to allow the client application to discover which of the API methods are supported.

In addition, the **info** method returns several information on the Remote Service which may be useful to a calling application to access its functions and features.

9 Authentication and authorization

This specification deals with two different and types of authentication and authorization:

- a. Service authentication and authorization.
- b. Credential authorization.

9.1 Service authorization and authentication

In order to protect the Remote Service from unauthorized access, this specification requires the signature application to obtain a valid “access token” to authorize the access to the APIs. This type of authorization is called service authorization. Various types of authorization mechanisms can be supported, and more will be supported in future versions, and the Signature Application shall adopt any of those available from the Remote Service as specified in response to the **info** method.

The Remote Service may also adopt an indirect way of authorizing access to the API. The underlying communication channel with the signature application may ensure access control in a different way, for example with a private point-to-point LAN connection or through a VPN (Virtual Private Network).

The access to the APIs shall also be authenticated. When the authentication is under the control of a Signature Application Provider, then the user shall be properly authenticated by this provider before getting access to the Remote Service. This scenario supports organizations that manage a user community with an existing form of authentication, for example a Bank managing the users from their Internet Banking service. This means that, in order to retrieve the signing credentials associated to a user, this organization

would have to take care of the correspondence between the user identifier in their own domain and the user identifier in the Remote Service's domain.

When the authentication is under the control of the Remote Service, the Signature Application shall perform a token-based authentication to the Remote Service by leveraging the security elements collected from the user, either through HTTP Basic or HTTP Digest authentication, or preferably via an OAuth 2.0 authorization mechanism. In practice, the Signature Application will require the user to authenticate directly to the Remote Service using any of the available methods. This would offer an authorization mechanism even in case the Signature Application and the Remote Service have not previously established any form of service authentication.

Two methods are defined in this specification to obtain an access token to authorize the access to the Remote Service API:

- The **auth/login** method shall be used when HTTP Basic or Digest authentication mechanism are preferred and supported by the Remote Service. The Signature Application will collect the security elements from the user and will pass them to the Remote Service.
- The **oauth2/token** method shall be used when an OAuth 2.0 authorization mechanism is preferred and supported by the Remote Service. The Signature Application will not collect any security elements from the user, but instead it will redirect to the Remote Service that will authenticate the user in its own web-based user interface. See Section 9.3 for further information on how to implement OAuth 2.0 authorization.

In both cases, if the user grants the authorization, the Remote Service will return a service access token to the Signature Application. From then on, all authorized requests to the API methods shall use an Authorization header with the type Bearer followed by this service access token.

If the user does not grant the permission, the Remote Service will return an error message.

9.2 Credential authorization

Accessing a credential for remote signing requires an authorization from the user who owns it to ensure the "sole control" of the signing key associated to it. Typically, the RSSP or the RSCD implement a Signature Activation Module (SAM) that control the key store (e.g. HSM) giving users the ability to authorize the access to their signing keys.

The SAM can manage the authorization in different ways, with different technologies and a variable number of authorization factors. This really depends on the implementation and on the policy adopted by the RSSP, and may also be determined by the level of compliance to industry and regulatory requirements, like in the case of standards like CEN EN 419 241-1, which defines different levels of "sole control assurance".

Three different types of credential authorization are defined and supported in this specification:

- Implicit authorization
- Explicit authorization
- OAuth 2.0 authorization

Implicit authorization means that the Remote Service is taking care of the authorization process autonomously, by engaging with the user without any intermediation from the Signature Application. In this case, the Signature Application will invoke the credential authorization methods without passing any security elements, as these would be implicitly managed by the Remote Service directly with the user.



Implicit authorization supports the SCAL1 (Sole Control Assurance Level 1) case defined in CEN 419 241-1 as well as SCAL2. In the first case, the service authorization activated with the security elements provided by the user is sufficient as a basic authorization of the signing key associated to the credential. With the SCAL 2 case, the SAM provides a completely independent two-factor authorization mechanism that does not require any user interaction to occur within the Signature Application.

Explicit authorization means that the Remote Service relies on the Signature Application to collect secure elements like static and/or dynamic one-time passwords from the user in its own environment, based on the parameters returned by the **credentials/info** method. This method returns the number, type and format of the required or optional secure elements, such that the Signature Application could show the proper interactive controls to collect them from the user.

A common type of explicit authorization used by a SAM is based on a static numeric PIN - generally defined by the user - associated to the signing key when it is generated. To increase the level of assurance of user control, ensuring that only the authorized user could create a signature with a certain credential, a stronger authorization factor may be adopted. A dynamically generated text-based OTP (One-Time Password) is a common strong authorization mechanism. PIN and OTP are supported directly in this specification and can be used in combination to service authorization to achieve the highest levels of assurance of the user's sole control, fully supporting the requirements for SCAL 1 and SCAL2 as defined in CEN 419 241-1.

Biometric validation and phone call drop are also examples of possible authorization mechanisms. These and other authorization mechanisms can be supported by means of an OAuth 2.0 authorization scheme.

9.3 OAuth 2.0 Authorization

OAuth 2.0 is an authorization framework that enables applications to obtain access to HTTP based services. It provides client applications a "secure delegated access" to server resources on behalf of a resource owner. This allows resource owners to authorize third-party access to their server resources without sharing their credentials.

Using the OAuth 2.0 authorization scheme, the Signature Application will show a web page managed by the Remote Service where the user will be authenticated according to the specific mechanism implemented there. After a successful authentication, the authorization server of the Remote Service will return an authorization code or an access token to the Signature Application. This access token will be used later to authorize access to the Remote Service's resources.

This specification supports the main types of OAuth 2.0 implementations as described in RFC 6749:

- Authorization Code flow
- Implicit Grant flow
- Client Credentials flow

OAuth 2.0 authorization mechanisms can be used for both Service and Credentials authorization, with the exception of the Client Credentials flow, which should be only used for Service authorization as it does not offer a form of user authentication. A Remote Service can therefore implement a single OAuth 2.0 authorization server supporting two different scopes for "service" and "credential".

Before using an OAuth 2.0 authorization mechanism, the Signature Application shall obtain from the Remote Service the client credentials (a Client ID and conditionally a Client Secret) and register one or more Redirect URI address with it. The means through which the Signature Application registers with the Remote Service are beyond the scope of this specification.



The following pages describe the OAuth 2.0 mechanisms supported by this specification and how to invoke them. Notice that the Client Credential flow is not described separately because it can be invoked by means of the **oauth2/token** method using a *grant_type* with value “client_credentials”.

PRELIMINARY RELEASE



OAuth 2.0 Authorization Code [`oauth2/authorize`]

Description: Starts the OAuth 2.0 authorization server using an Authorization Code flow, as described in Section 1.3.1 of RFC 6749, to request authorization for the user to access the Remote Service resources. The authorization is returned in the form of an authorization code, which the Signature Application shall then use to obtain an access token with the **oauth2/token** method. The authorization server should support two access token scopes: “service” and “credential”. These scopes shall be used to obtain an access token suitable for service and credentials authorization respectively.

NOTE: **oauth2/authorize** does not specify a regular API method, but rather the recommended URI fragment of the address of the web page allowing the user sign-in to the Remote Service to authorize the Signature Application. The complete URL that shall be used for invoking the authorization server is returned in the *oauth2* parameter of the **info** method, and does not necessarily include the default base URI of the Remote Service API. At the end of the authorization process, the authorization server shall redirect the user-agent by sending the HTTP/1.1 302 Found response with a Location header containing the URI specified by the *redirect_uri* parameter, which shall be pre-registered with the Remote Service by the Signature Application.

Input: In case the scope of the OAuth 2.0 authorization request is “credential”, the Bearer service token shall be added to the Authorization header. In order to maintain compatibility with the OAuth 2.0 standard, the following parameters shall be passed as URL-encoded query parameters, and not as JSON data in the body of the request.

Parameter	Presence	Value	Description
<i>response_type</i>	Required	<i>String</i> code	The value shall be “code”.
<i>client_id</i>	Required	<i>String</i>	This is the unique “client ID” previously assigned to the Signature Application by the Remote Service.
<i>redirect_uri</i>	Optional	<i>String</i>	The URL where the user will be redirected after the authorization process has completed. Only a valid URI pre-registered with the Remote Service shall be passed. If omitted, the Remote Service will use the default redirect URI pre-registered by the Signature Application.
<i>scope</i>	Optional	<i>String</i> service credential	The scope of the access request as described by Section 3.3 of RFC 6749. <ul style="list-style-type: none">“service”: it shall be used to obtain an access token suitable for service authorization.“credential”: it shall be used to obtain an access token suitable for credentials authorization. The parameter is optional. The defaults scope is “service” in case it is omitted.
<i>state</i>	Optional	<i>String</i>	Up to 255 bytes of arbitrary data from the Signature Application that will be passed back to the redirect URI. It can be used to handle a transaction identifier or other application-specific data. The use is recommended for preventing cross-site request forgery.
<i>lang</i>	Optional	<i>String</i>	Request a preferred language according to RFC 3066. If specified, the Remote Service should render the authorization web page in this language, if supported. If omitted, the Remote Service shall render the authorization web page in its own default language.



<i>credentialID</i>	Required Conditional	<i>String</i>	The identifier associated to the credential to authorize. It shall be used only if the scope of the OAuth 2.0 authorization request is "credential".
<i>numSignatures</i>	Required Conditional	<i>Number</i>	The number of signatures to authorize. Multi-signature transactions can be obtained by using a combination of array of hash values and by calling the signatures/signHash method multiple times. It shall be used only if the scope of the OAuth 2.0 authorization request is "credential".
<i>hash</i>	Required Conditional	<i>Array of String</i>	One or more Base64-encoded hash values to be signed. It shall be used only if the scope of the OAuth 2.0 authorization request is "credential" and the SCAL parameter returned by credentials/info is "2".

Output: After a successful user authentication, the authorization server shall redirect the user-agent by sending the HTTP/1.1 302 Found response with a Location header containing the URI specified by the *redirect_uri* parameter with the following URL-encoded query parameters.

Attribute	Presence	Value	Description
<i>code</i>	Required	<i>String</i>	The authorization code generated by the authorization server. It shall be bound to the client identifier and the redirection URI. It shall expire shortly after it is issued to mitigate the risk of leaks. The Signature Application cannot use the value more than once.
<i>state</i>	Conditional	<i>String</i>	Contains the arbitrary data from the signature application that was specified in the input request. It shall be returned only when specified in the request.
<i>error</i>	Conditional	<i>String</i> invalid_request unauthorized_client access_denied unsupported_response_type invalid_scope server_error temporarily_unavailable	A single error code string from the following list: <ul style="list-style-type: none">• "invalid_request": it shall be used if the request is missing a required parameter.• "unauthorized_client": it shall be used if the client is not authorized.• "access_denied": it shall be used if the server denied the request.• "unsupported_response_type": it shall be used if the server does not support the required response type.• "invalid_scope": it shall be used if the requested scope is invalid, unknown, or malformed.• "server_error": it shall be used if the server encountered an unexpected condition that prevented it from fulfilling the request.• "temporarily_unavailable": it shall be used if the server is currently unable to handle the request due to temporary overloading or maintenance.
<i>error_description</i>	Conditional	<i>String</i>	Human-readable text providing additional error information.
<i>error_uri</i>	Conditional	<i>String</i>	A URI identifying a human-readable web page with information about the error.

Sample Request (service authorization)

GET [https://www.domain.org/oauth2/authorize?response_type=code&client_id=\[CLIENT_ID\]&redirect_uri=\[REDIRECT_URI\]&scope=service&state=12345678&lang=en-US&appName=Cloud%20Signature%20Service](https://www.domain.org/oauth2/authorize?response_type=code&client_id=[CLIENT_ID]&redirect_uri=[REDIRECT_URI]&scope=service&state=12345678&lang=en-US&appName=Cloud%20Signature%20Service)

Sample Response



HTTP/1.1 302 Found
Location: [REDIRECT_URI]?code=FhkXf9P269L8g&state=12345678

Sample Request (credential authorization)

GET [https://www.domain.org/oauth2/authorize?response_type=code&client_id=\[CLIENT_ID\]&redirect_uri=\[REDIRECT_URI\]&scope=credential&state=12345678&credentialID=GX0112348&numSignatures=1](https://www.domain.org/oauth2/authorize?response_type=code&client_id=[CLIENT_ID]&redirect_uri=[REDIRECT_URI]&scope=credential&state=12345678&credentialID=GX0112348&numSignatures=1)
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

Sample Response

HTTP/1.1 302 Found
Location: [REDIRECT_URI]?code=HS9naJKWwp901hBcK348IUHiuH8374&state=12345678

PRELIMINARY RELEASE



OAuth 2.0 Implicit Grant [`oauth2/authorize`]

Description: Starts the OAuth 2.0 authorization server using an Implicit grant flow, as described in Section 1.3.2 of RFC 6749, to request authorization for the user to access the Remote Service resources. The authorization is returned in the form of an access token that the Signature Application shall use directly to invoke the **signatures/signHash** method. This method is a simplified authorization code flow in which the Signature Application is issued an access token directly. The authorization server should support two access token scopes: “service” and “credential”. These scopes shall be used to obtain an access token suitable for service and credentials authorization respectively.

NOTE: `oauth2/authorize` does not specify a regular API method, but rather the recommended URI fragment of the address of the web page allowing the user sign-in to the Remote Service to authorize the Signature Application. The complete URL that shall be used for invoking the authorization server is returned in the `oauth2` parameter of the **info** method, and does not necessarily include the default base URI of the Remote Service API. At the end of the authorization process, the authorization server redirects the user-agent by sending the HTTP/1.1 302 Found response with a Location header containing the URI specified by the `redirect_uri` parameter, which shall be pre-registered with the Remote Service by the Signature Application.

Input: In case the scope of the OAuth 2.0 authorization request is “credential”, the Bearer service token shall be added to the Authorization header. In order to maintain compatibility with the OAuth 2.0 standard, the following parameters shall be passed as URL-encoded query parameters, and not as JSON data in the body of the request.

Parameter	Presence	Value	Description
<code>response_type</code>	Required	<i>String</i> Token	The value shall be “token”.
<code>client_id</code>	Required	<i>String</i>	This is the unique “client ID” previously assigned to the Signature Application by the Remote Service.
<code>redirect_uri</code>	Optional	<i>String</i>	The URL where the user will be redirected after the authorization process has completed. Only a valid URI pre-registered with the Remote Service shall be passed. If omitted, the Remote Service will use the default redirect URI pre-registered by the Signature Application.
<code>scope</code>	Optional	<i>String</i> service credential	The scope of the access request as described by Section 3.3 of RFC 6749. <ul style="list-style-type: none">“service”: it shall be used to obtain an access token suitable for service authorization.“credential”: it shall be used to obtain an access token suitable for credentials authorization. The parameter is optional. The defaults scope is “service” in case it is omitted.
<code>state</code>	Optional	<i>String</i>	Up to 255 bytes of arbitrary data from the Signature Application that will be passed back to the redirect URI. It can be used to handle a transaction identifier or other application-specific data. The use is recommended to prevent cross-site request forgery.
<code>lang</code>	Optional	<i>String</i>	Request a preferred language according to RFC 3066. If specified, the Remote Service should render the authorization web page in this language, if supported. If omitted, the Remote Service shall render the authorization web page in its own default language.



<i>credentialID</i>	Required Conditional	<i>String</i>	The identifier associated to the credential to authorize. It shall be used only if the scope of the OAuth 2.0 authorization request is "credential".
<i>numSignatures</i>	Required Conditional	<i>Number</i>	The number of signatures to authorize. Multi-signature transactions can be obtained by using a combination of array of hash values and by calling the signatures/signHash method multiple times. It shall be used only if the scope of the OAuth 2.0 authorization request is "credential".
<i>hash</i>	Required Conditional	<i>Array of String</i>	One or more Base64-encoded hash values to be signed. It shall be used only if the scope of the OAuth 2.0 authorization request is "credential" and the SCAL parameter returned by credentials/info is "2".

Output: After a successful user authentication, the authorization server redirects the user-agent by sending the HTTP/1.1 302 Found response with a Location header containing the URI specified by the *redirect_uri* parameter with the following URL-encoded query parameters.

Attribute	Presence	Value	Description
<i>access_token</i>	Required	<i>String</i>	The short-lived service access token. It shall be used depending on the scope of the OAuth 2.0 authorization request. When the scope is "service" then it shall be used as the value of the "Authorization: Bearer" in the HTTP header of the subsequent API requests within the same session. When the scope is "credential" then it shall be used as a Signature Activation Data token to authorize the signature request. This value shall be used as the value for the SAD parameter when invoking the signatures/signHash method.
<i>token_type</i>	Required	<i>String</i> Bearer	Specifies a "Bearer" token type as defined in RFC 6750.
<i>expires_in</i>	Optional	<i>Number</i>	The lifetime in seconds of the service access token. If omitted, the default expiration time is 3600 (1 hour).
<i>state</i>	Conditional	<i>String</i>	Contains the arbitrary data from the signature application that was specified in the input request. It shall be returned only when specified in the request.
<i>error</i>	Conditional	<i>String</i> invalid_request unauthorized_client access_denied unsupported_response_type invalid_scope server_error temporarily_unavailable	A single error code string from the following list: <ul style="list-style-type: none">• "invalid_request": it shall be used if the request is missing a required parameter.• "unauthorized_client": it shall be used if the client is not authorized.• "access_denied": it shall be used if the server denied the request.• "unsupported_response_type": it shall be used if the server does not support the required response type.• "invalid_scope": it shall be used if the requested scope is invalid, unknown, or malformed.• "server_error": it shall be used if the server encountered an unexpected condition that prevented it from fulfilling the request.• "temporarily_unavailable": it shall be used if the server is currently unable to handle the request due to temporary overloading or maintenance.
<i>error_description</i>	Conditional	<i>String</i>	Human-readable text providing additional error information.
<i>error_uri</i>	Conditional	<i>String</i>	A URI identifying a human-readable web page with information about the error.



Sample Request (service authorization)

GET [https://www.domain.org/oauth2/authorize?response_type=token&client_id=\[CLIENT_ID\]&redirect_uri=\[REDIRECT_URI\]&scope=service&state=12345678&lang=en-US&appName=Cloud%20Signature%20Service](https://www.domain.org/oauth2/authorize?response_type=token&client_id=[CLIENT_ID]&redirect_uri=[REDIRECT_URI]&scope=service&state=12345678&lang=en-US&appName=Cloud%20Signature%20Service)

Sample Response

HTTP/1.1 302 Found
Location: [REDIRECT_URI]?access_token=FhkXf9P269L8g&token_type=Bearer&expires_in=3600&state=12345678

Sample Request (credential authorization)

GET [https://www.domain.org/oauth2/authorize?response_type=token&client_id=\[CLIENT_ID\]&redirect_uri=\[REDIRECT_URI\]&scope=credential&state=12345678&credentialID=GX0112348&numSignatures=1](https://www.domain.org/oauth2/authorize?response_type=token&client_id=[CLIENT_ID]&redirect_uri=[REDIRECT_URI]&scope=credential&state=12345678&credentialID=GX0112348&numSignatures=1)
Authorization: Bearer 4/CKN69L8gdSYp5_pwh3XlFQZ3ndFhkXf9P2_TiHRG-bA

Sample Response

HTTP/1.1 302 Found
Location: [REDIRECT_URI]?access_token=FhkXf9P269L8g&token_type=Bearer&expires_in=3600state=12345678



10 Creating a Remote Signature

Remote Signature services allow generating digital signatures remotely by means of a RSCD operated as a service. A Remote Signature Service Provider is an organization that manages the RSCD on behalf of the signers.

The RSCD should be able to manage the following remote signature scenarios:

- The remote signature of a single hash;
- The remote signature of multiple hashes passed in a single signature operation;
- The remote signature of multiple hashes passed across multiple signature operations occurring within a single session.

In general, each time a remote signature is required, the strong authentication mechanism should be invoked. However, in order to improve the signer's experience, the strong authentication may be allowed to occur only once per signing session covering multiple signatures. This possibility depends on the policy of the Remote Service, which may not allow it, based on regulatory or security requirements.

See Section 13 to understand all the workflows supported in this specification and the sequence of API calls to be invoked in order to create the supported types of remote signatures.

10.1 Multi-signature Transactions

This specification requires credentials authorization every time a signature is created. This means that applying multiple signatures may require the user to authorize the Signature Application multiple times in a rapid sequence, which would be cumbersome in case of strong authorization mechanisms like OTP. To simplify processing a set of signatures or documents, the Remote Service may support the ability to authorize multi-signature transactions.

A multi-signature transaction allows a user to sign multiple times although requiring a single authorization assertion (for example a single OTP).

A multi-signature transaction can be created by submitting multiple hash values at once (suitable for batch signing) or by invoking the **signatures/signHash** method multiple times. In both cases, the authorization mechanism adopted by the Signature Application shall explicitly specify the total number of signatures to be authorized.

The ability to create a multi-signature transaction depends on the policy adopted by the Remote Service. The *multisign* parameter of the **credentials/info** method should be used to check if multi-signature transactions are supported or not by a specific credential.

11 Error handling

Errors are returned by the Remote Service using standard HTTP status code syntax. Any additional info is included as JSON in the body of the response from an API request.

The HTTP protocol defines a list of standard status codes that are referenced in this document to help the Signature Application deal with these responses accordingly. A Remote Service conforming to this specification should support all the following HTTP status codes:



Table 1 – HTTP Status Codes

Standard Status Code	Description
200 OK	Response to a successful API method request.
302 Found	Response used to redirect the user to an OAuth 2.0 authorization endpoint.
400 Bad Request	Returned when an error is returned due to an unsupported or invalid parameters or missing required parameters.
401 Unauthorized	Returned when a bad or expired authorization token is used.
429 Too Many Requests	Returned when a request is rejected due to rate limiting.
500 Internal Server Error	Returned when the server encounters an unexpected condition.
501 Not Implemented	Returned when an unimplemented method is requested.
503 Service Unavailable	Returned when the server is currently unable to handle the request due to temporary overloading or maintenance conditions.

11.1 Error messages

Just as an HTML error page shows a useful error message to a visitor, the Remote Service implementing the API described in this specification shall provide a useful error message in case something goes wrong. When an error is detected, the Remote Service shall return an HTTP Status Code 400 Bad Request and the information on the error shall be returned by the Remote Service in the body of the HTTP response using the "application/json" media type as defined by RFC 4627. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings as per the following example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
{
  "error": "invalid_request",
  "error_description": "This is not a valid access token"
}
```

The *error_description* parameter is Optional but highly recommended to contain a human-readable text string providing additional information to assist the Signature Application in understanding the error that occurred.

The Remote Service can define custom error messages by using messages that are not defined in this specification. Notice that these error messages may have an inconsistent meaning among different implementations of this specification.

Error Codes from the following predefined list have a very precise meaning according to the corresponding Error Description:

Table 2 – Predefined Error Messages

Error	Error Description
invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
unauthorized_client	The client is not authorized to use this method.
access_denied	The user, authorization server or Remote Service denied the request.
unsupported_response_type	The authorization server does not support obtaining an authorization code using this method.
invalid_scope	The requested scope is invalid, unknown, or malformed.



server_error	The authorization server encountered an unexpected condition that prevented it from fulfilling the request.
temporarily_unavailable	The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server.
expired_token	The access or refresh token is expired or has been revoked.
invalid_token	The token provided is not a valid OAuth access or refresh token.

12 The Remote Service APIs

In order to simplify the navigation of this specification, the following table summarizes all the API methods defined in the present document.

Table 3 – API methods summary

URI	Description
info	Returns information on the Remote Service and the list of API methods it has implemented.
auth/login	Authorize the Remote Service with HTTP Basic or Digest authentication.
auth/revoke	Revoke the service access token or refresh token.
oauth2/token	Obtain an OAuth 2.0 access token.
credentials/list	Returns the list of credentials associated to a user.
credentials/info	Returns information on a signing credential, its associated certificate and a description of the supported authorization mechanism.
credentials/authorize	Authorize the access to the credential for signing.
credentials/extendTransaction	Extend the validity of a multi-signature transaction.
credentials/sendOTP	Start the online OTP mechanism associated to a credential.
signatures/signHash	Calculate a raw digital signature from one or more hash values.
signatures/timestamp	Return a time stamp token for the input hash value.

**info**

Description: Returns several information about the Remote Service and the list of the API methods implemented and supported by it.

Input: This method allows the following parameters:

Parameter	Presence	Value	Description
<i>lang</i>	Optional	<i>String</i>	Request a preferred language of the response to the Remote Service, specified according to RFC 3066. If present, the Remote Service shall provide language-specific responses using the specified language. If the specified language is not supported then it shall provide these responses in the language as specified in the <i>lang</i> output parameter.

Output: This method returns the following parameters.

Attribute	Presence	Value	Description
<i>specs</i>	Required	<i>String</i>	The version of this specification implemented by the provider. The format of the string is Major.Minor.x.y, where Major is a number equivalent to the API version (e.g. 1 for API v1) and Minor is a number identifying the version update, while x and y are subversion numbers.
<i>name</i>	Required	<i>String</i>	The commercial name of the Remote Service.
<i>logo</i>	Required	<i>String</i>	The URI of the image file containing the logo of the Remote Service which shall be published online. The image shall be in either JPEG or PNG format and not larger than 256x256 pixels.
<i>region</i>	Required	<i>String</i>	The ISO 3166-1 Alpha-2 code of the Country where the Remote Service provider is established (e.g. ES for Spain).
<i>lang</i>	Required	<i>String</i>	The language used in the responses, specified according to RFC 3066.
<i>description</i>	Required	<i>String</i>	Contains a free form description of the Remote Service in the <i>lang</i> language. The maximum size of the string shall be 255 characters.
<i>authType</i>	Required	<i>Array of String</i>	Specifies one or more values corresponding to the authentication mechanisms supported by the Remote Service to authorize the access to the API: <ul style="list-style-type: none">• “external”: in case the authorization is managed externally (e.g. using a VPN or a private LAN).• “TLS”: in case the authorization is provided by means of TLS client certificate authentication.• “basic”: in case of HTTP Basic Authentication.• “digest”: in case of HTTP Digest Authentication.• “oauth2code”: in case of OAuth 2.0 with authorization code flow.• “oauth2implicit”: in case of OAuth 2.0 with implicit grant flow.• “oauth2client”: in case of OAuth 2.0 with client credentials flow.
<i>oauth2</i>	Required Conditional	<i>String</i>	Specifies the complete URI of the OAuth 2.0 service authorization endpoint provided by the Remote Service. The parameter is required only if the <i>authType</i> parameter contains “oauth2code” or “oauth2implicit”.
<i>methods</i>	Required	<i>Array of String</i>	Specifies the list of names of all the API methods described in this specification that are implemented and supported by the Remote Service.



Sample Request

POST <https://service.domain.org/csc/v0/info>

Sample Response

HTTP/1.1 200 OK

```
{
  "specs": "1.1",
  "name": "ACME Trust Services",
  "logo": "https://service.domain.org/images/logo.png",
  "region": "IT",
  "lang": "en-US",
  "authType": ["basic", "oauth2code", "oauth2implicit"],
  "oauth2": "https://www.domain.org/oauth2/authorize",
  "methods":
  [
    "auth/login",
    "auth/revoke",
    "oauth2/token",
    "credentials/list",
    "credentials/info",
    "credentials/authorize",
    "credentials/sendOTP",
    "signatures/signHash"
  ]
}
```



auth/login

Description: Obtain an access token for service authorization from the Remote Service using HTTP Basic Authentication or HTTP Digest authentication, as defined in RFC 2617, using the “*userID*” and “*password*” assigned to the user. These secure elements shall be passed directly in the HTTP header as an authorization grant to obtain a service access token to use for the subsequent API requests within the same session.

The *rememberMe* parameter can be optionally used, under the control of the user, in order to extend a successful authentication for the subsequent session, and avoid the user to authenticate again within a predefined period of time. In this case a refresh token will be returned, which can be passed in the *refresh_token* parameter in subsequent calls as an alternative to passing “*userID*” and “*password*” to obtain a new access token.

NOTE: HTTP Basic Authentication is not a completely safe mechanism and therefore it is not recommended for use, especially by Signature Application Providers. This method might also be deprecated in future releases of this specification. The recommended mechanism for user authentication is OAuth 2.0 (see Section 9.3).

Input: The “*userID*” and “*password*” strings shall be encoded as defined in RFC 2617 and provided into the Authorization HTTP header. Alternatively, a refresh token can be used to re-authenticate the user after the access token has expired. This method allows the following parameters:

Parameter	Presence	Value	Description
<i>refresh_token</i>	Required Conditional	<i>String</i>	The long-lived refresh token returned from the previous HTTP Basic Authentication. This is used as an alternative to passing the Authorization header to reauthenticate the user according to the method described in RFC 6749 par. 1.5. NOTE: This refresh token is not compatible with refresh tokens obtained by means of OAuth 2.0 authorization.
<i>rememberMe</i>	Optional	<i>Boolean</i>	This parameter normally corresponds to an option that the user may activate during the authentication phase to “stay signed in” and maintain the authentication valid across multiple sessions. <ul style="list-style-type: none">“true”: if the Remote Service supports user reauthentication, a <i>refreshToken</i> will be returned and the signature application may use it on the subsequent session instead of passing the Authorization header.“false”: if the Remote Service does not support user reauthentication, a <i>refresh_token</i> will not be returned. If the parameter is omitted, it will default to “false”. This mechanism is based on the method described in RFC 6749 par. 1.5.
<i>clientData</i>	Optional	<i>String</i>	Arbitrary data from the Signature Application. It can be used to handle a transaction identifier or other application-specific data.

Output value: This method returns the following parameters.



Attribute	Presence	Value	Description
<i>access_token</i>	Required	<i>String</i>	The short-lived service access token used to authenticate the subsequent API requests within the same session. This shall be used as the value of the "Authorization: Bearer" in the HTTP header of the API requests. The access token has a limited time validity. In case of expired token, the Remote Service returns an error and a new auth/login request will be required.
<i>refresh_token</i>	Optional Conditional	<i>String</i>	The long-lived refresh token used to re-authenticate the user on the subsequent session. The value is returned if the <i>rememberMe</i> parameter in the request is "true" and the Remote Service supports user reauthentication. This mechanism is based on the method described in RFC 6749 par. 1.5. NOTE: This refresh_token is not compatible with refresh tokens obtained by means of OAuth 2.0 authorization.
<i>expires_in</i>	Optional	<i>Number</i>	The lifetime in seconds of the service access token. If omitted, the default expiration time is 3600 (1 hour).

Error Case	Status Code	Error	Error Description
The authorization header does not match the basic HTTP authentication pattern ("Basic [base64]") - if refresh token is not present	401 (unauthorized)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
Decoded credentials are not in the form "username:password"	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
Invalid refresh_token parameter format	400 (bad request)	invalid_request	Invalid string parameter: refresh_token
Invalid refresh_token value	400 (bad request)	invalid_request	Invalid refresh_token
Authentication error – login failed	400 (bad request)	authentication_error	An error occurred during authentication process

Sample Request

```
POST https://service.domain.org/csc/v0/auth/login
Authorization: Basic Y2xpZW50X2lkOmNsaWVudF9zZWNYZXQ=
Content-Type: application/json
{
  "rememberMe": true
}
```

Sample Response

```
HTTP/1.1 200 OK
{
  "access_token": "4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA",
  "refresh_token": "_TiHRG-bA_H3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
  "expires_in": 3600
}
```



auth/revoke

Description: Revoke a service access token or refresh token that was obtained from the Remote Service, as described in RFC 7009. This method exists to enforce the security of the Remote Service. When the Signature Application needs to terminate a session, it is recommended to invoke this method to prevent further access by reusing the token. This method allows the Signature Application to invalidate its tokens according to the following approach: If the token passed to the request is a refresh token, then the authorization server shall also invalidate all access tokens based on the same authorization grant. If the token passed to the request is an access token, then the server shall not revoke any existing refresh token based on the same authorization grant. The invalidation of the token takes place immediately, and the token cannot be used again after its revocation.

Input: This method allows the following parameters:

Parameter	Presence	Value	Description
<i>token</i>	Required	<i>String</i>	The token that the Signature Application wants to get revoked.
<i>token_type_hint</i>	Optional	<i>String</i> access_token refresh_token	Specifies an optional hint about the type of the token submitted for revocation. If the parameter is omitted, the authorization server should try to identify the token across all the available tokens.
<i>clientData</i>	Optional	<i>String</i>	Arbitrary data from the Signature Application. It can be used to handle a transaction identifier or other application-specific data.

Output: This method has no output parameters.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
Missing or not String "token" parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter token
"token_hint" parameter present, not equal to "access_token" nor "refresh_token"	400 (bad request)	invalid_request	Invalid string parameter token_type_hint
Invalid access_token or refresh_token	400 (bad request)	invalid_request	Invalid string parameter token

Sample Request

```
POST https://service.domain.org/csc/v0/auth/revoke
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json
{
  "token": "_TiHRG-bA H3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
  "token_type_hint": "refresh_token",
  "clientData": "12345678"
}
```



Sample Response

HTTP/1.1 200 OK

PRELIMINARY RELEASE



oauth2/token

Description: Obtain an OAuth 2.0 bearer access token from the Remote Service by passing either the client credentials pre-assigned by the Remote Service to the Signature Application, or the authorization code or refresh token returned by the Authorization Server after a successful user authentication, along with the client ID and client secret in possession of the Signature Application. This method shall be used only in case of an Authorization Code flow as described in Section 1.3.1 of RFC 6749, in case of Client Credential flow as described in Section 1.3.4 of RFC 6749 or in case of Refresh Token flow as described in Section 1.5 of RFC 6749. Notice that the Client Credential flow and Refresh Token flow can be used only for service authorization.

Input: In case the scope of the OAuth 2.0 authorization request is “credential”, the Bearer service token shall be added to the Authorization header. This method allows the following parameters:

Parameter	Presence	Value	Description
<i>grant_type</i>	Required	<i>String</i> authorization_code client_credentials refresh_token	The grant type, which depends on the type of OAuth 2.0 flow: <ul style="list-style-type: none">“authorization_code”: shall be used in case of Authorization Code Grant.“client_credentials”: shall be used in case of Client Credentials Grant.“refresh_token”: shall be used in case of Refresh Token flow.
<i>code</i>	Required Conditional	<i>String</i>	The authorization code returned by the authorization server. It shall be bound to the client identifier and the redirection URI. This shall be used only when <i>grant_type</i> is “authorization_code”.
<i>refresh_token</i>	Required Conditional	<i>String</i>	The long-lived refresh token returned from the previous session. This shall be used only when the scope of the OAuth 2.0 authorization request is “service” and <i>grant_type</i> is “refresh_token” to reauthenticate the user according to the method described in Section 1.5 of RFC 6749.
<i>client_id</i>	Required	<i>String</i>	This is the unique “client ID” previously assigned to the Signature Application by the Remote Service.
<i>client_secret</i>	Required	<i>String</i>	This is the “client secret” previously assigned to the Signature Application by the Remote Service.
<i>redirect_uri</i>	Required Conditional	<i>String</i>	The URL where the user was redirected after the authorization process completed. It is used to validate that it matches the original value previously passed to the Authorization Server. This shall be used only if the <i>redirect_uri</i> parameter was included in the authorization request, and their values shall be identical.
<i>clientData</i>	Optional	<i>String</i>	Arbitrary data from the Signature Application. It can be used to handle a transaction identifier or other application-specific data.

Output value: This method returns the following parameters:

Attribute	Presence	Value	Description
<i>access_token</i>	Required	<i>String</i>	The short-lived access token to be used depending on the <i>scope</i> of the OAuth 2.0 authorization request. When the scope is “service” then the Authorization Server returns a bearer token to be used as the value of



			the "Authorization: Bearer" in the HTTP header of the subsequent API requests within the same session. When the scope is "credential" then the Authorization Server returns a Signature Activation Data token to authorize the signature request. This value should be used as the value for the SAD parameter when invoking the signatures/signHash method.
<i>refresh_token</i>	Optional	<i>String</i>	The long-lived refresh token used to re-authenticate the user on the subsequent session based on the method described in Section 1.5 of RFC 6749. The presence of this parameter is controlled by the user and is allowed only when the scope of the OAuth 2.0 authorization request is "service".
<i>token_type</i>	Required	<i>String</i> Bearer	Specifies a "Bearer" token type as defined in RFC6750.
<i>expires_in</i>	Optional	<i>Number</i>	The lifetime in seconds of the service access token. If omitted, the default expiration time is 3600 (1 hour).

Error Case	Status Code	Error	Error Description
Missing or not String "client_id" parameter	400 (bad request)	invalid_request	parameter [client_id] cannot be empty
Missing or not String "client_secret" parameter	400 (bad request)	invalid_request	parameter [client_secret] cannot be empty
Missing or not String "grant_type" parameter	400 (bad request)	invalid_request	parameter [grant_type] cannot be empty
Invalid parameter "grant_type"	400 (bad request)	invalid_request	Invalid parameter grant_type
Missing or not String "code" parameter	400 (bad request)	invalid_request	parameter [code] cannot be empty
Missing or not String "refresh_token" parameter	400 (bad request)	invalid_request	parameter [refresh_token] cannot be empty
Invalid "client_id" parameter	400 (bad request)	invalid_request	Invalid parameter client_id
Invalid "client_secret" parameter	400 (bad request)	invalid_request	Invalid parameter client_secret
The "redirect_uri" parameter does not match any of the client_id registered redirect_uri regex	400 (bad request)	invalid_request	redirect_uri parameter not allowed
Expired "access_token"	400 (bad request)	invalid_request	Session expired
The authorization header does not match the pattern "Bearer [sessionKey]"	401 (unauthorized)	invalid_request	Missing access_token
Missing access_token	400 (bad request)	access_denied	Missing access_token
Invalid access_token	400 (bad request)	access_denied	Invalid access_token
Expired "SAD"	400 (bad request)	invalid_request	SAD expired
Invalid "refresh_token" parameter	400 (bad request)	invalid_request	Invalid parameter refresh_token
Authorization code expired	400 (bad request)	expired_token	Authorization code expired

Sample Request (authorization code)

POST <https://service.domain.org/csc/v0/oauth2/token>



```
Content-Type: application/json
{
  "grant_type": "authorization_code",
  "code": "FhkXf9P269L8g",
  "client_id": "test",
  "client_secret": "password"
}
```

Sample Response

```
HTTP/1.1 200 OK
{
  "access_token": "4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA",
  "refresh_token": "_TiHRG-bA H3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Sample Request (refresh token)

```
POST https://service.domain.org/csc/v0/oauth2/token
Content-Type: application/json
{
  "grant_type": "refresh_token",
  "refresh_token": "_TiHRG-bA H3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
  "client_id": "test",
  "client_secret": "password"
}
```

Sample Response

```
HTTP/1.1 200 OK
{
  "access_token": "4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA",
  "refresh_token": "HfoOIhOIH/D8huiygIH98h_8hGH9iIUASDfKk8v98YHSDa",
  "token_type": "Bearer",
  "expires_in": 3600
}
```



credentials/list

Description: Returns the list of credentials associated with a user identifier. A user may have one or multiple credentials associated within a single Remote Signature Service Provider. If the user is authenticated directly by the RSSP then the *userID* is implicit and shall not be specified.

This method can also be used in case of a community of users, to get the list of credentials assigned to a particular user or to get the list of credentials assigned to the community. In this case the *userID* shall be passed explicitly to get the list of a specific user, or omitted to get the full list of users from the community.

Input: This method allows the following parameters:

Parameter	Presence	Value	Description
<i>userID</i>	Required Conditional	<i>String</i>	The user identifier associated to the user identity. This parameter shall be specified only when there is no user-specific authorization (e.g. when the <i>authType</i> returned by the <i>info</i> method is "external" or "TLS"). If the service authorization is user-specific (e.g. when the <i>authType</i> returned by the <i>info</i> method is "basic", "digest" or "oauth2...") the <i>userID</i> is already implicit in the service access token passed in the Authorization header. In this case, it shall not be possible to specify a different <i>userID</i> to obtain the list of credentials associated to another user, and the Remote Service shall return an error.
<i>maxResults</i>	Optional	<i>Number</i>	Maximum number of items to return. In case this parameter is omitted or invalid (value is too big) the Remote Service should return a predefined maximum number of items.
<i>pageToken</i>	Optional	<i>String</i>	The page token for the new page of items. The parameter is only required to retrieve results other than the first page.
<i>clientData</i>	Optional	<i>String</i>	Arbitrary data from the Signature Application. It can be used to handle a transaction identifier or other application-specific data.

Output: This method returns the following parameters:

Attribute	Presence	Value	Description
<i>credentialIDs</i>	Required	<i>Array of String</i>	One or more credentialID associated with the provided or implicit <i>userID</i> .
<i>nextPageToken</i>	Optional	<i>String</i>	The page token for the next page of items. No value is returned if the Remote Service does not supports items pagination or in case the last page is returned.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
"maxResults" < 1 or "maxResults" > 300	400 (bad request)	invalid_request	Invalid parameter maxResults
Invalid "pageToken" string format (not numeric)	400 (bad request)	invalid_request	Invalid parameter pageToken



Not empty "userID" parameter in case of user-specific authorization	400 (bad request)	invalid_request	userID parameter must be null
Invalid "userID" format in case of no user-specific authorization	400 (bad request)	invalid_request	Invalid parameter userID

Sample Request

POST <https://service.domain.org/csc/v0/credentials/list>

Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA

Sample Response

HTTP/1.1 200 OK

```
{
  "credentialIDs":
  [
    "GX0112348",
    "HX0224685"
  ]
}
```

PRELIMINARY RELEASE



credentials/info

Description: Retrieve the credential and return the main identity information and the public key certificate or the certificate chain associated to it. It can also return information about the authorization mechanism required to authorize or authenticate the access to the credential for remote signing, if requested.

Input: This method allows the following parameters:

Parameter	Presence	Value	Description
<i>credentialID</i>	Required	<i>String</i>	The identifier associated to the credential.
<i>certificates</i>	Optional	<i>String</i> none single chain	Specifies which certificates from the certificate chain shall be returned in <i>certs/certificates</i> . <ul style="list-style-type: none">• “none”: no certificate is returned.• “single”: only the end entity certificate is returned.• “chain”: the full certificate chain is returned. The default value is “single”, so if the parameter is omitted then the method will only return the end entity certificate.
<i>certInfo</i>	Optional	<i>Boolean</i>	Specifies if the information on the end entity certificate shall be returned as printable strings. This is useful in case the Signature Application wants to retrieve some details of the certificate without having to decode it. The default value is “false”, so if the parameter is omitted then the information will not be returned.
<i>authInfo</i>	Optional	<i>Boolean</i>	Specifies if the information on the authorization mechanisms supported by this credential (PIN and OTP groups) shall be returned. The default value is “false”, so if the parameter is omitted then the information will not be returned.
<i>lang</i>	Optional	<i>String</i>	Request a preferred language according to RFC 3066. If specified, the Remote Service should return the label and ddescription parameters in this language, if supported. If the language is not supported by the Remote Service, it should specify the default language in the output <i>lang</i> parameter. If omitted, the Remote Service should return these parameters in its own default language.
<i>clientData</i>	Optional	<i>String</i>	Arbitrary data from the Signature Application. It can be used to handle a transaction identifier or other application-specific data.

Output: This method returns the following parameters:

Attribute	Presence	Value	Description
<i>description</i>	Optional	<i>String</i>	A free form description of the credential in the <i>lang</i> language. The maximum size of the string is 255 characters.
<i>key/status</i>	Required	<i>String</i> enabled disabled	The status of enablement of the signing key of the credential: <ul style="list-style-type: none">• “enabled”: the signing key is enabled and can be used for signing.• “disabled”: the signing key is disabled and cannot be used for signing. This may occur when the owner has disabled it or when the RSSP has detected that the associated certificate is expired or revoked.



<i>key/algo</i>	Required	<i>Array of String</i>	The list of OIDs of the supported key algorithms. For example: 1.2.840.113549.1.1.1 = RSA encryption, 1.2.840.10045.4.3.2 = ECDSA with SHA256.
<i>key/len</i>	Required	<i>Number</i>	The length of the cryptographic key in bits.
<i>key/curve</i>	Required Conditional	<i>String</i>	The OID of the ECDSA curve. The value shall only be returned if <i>keyAlgo</i> is based on ECDSA.
<i>cert/status</i>	Optional	<i>String</i> valid expired revoked suspended	The status of validity of the end entity certificate. The value is optional, so the Remote Service should only return a value that is accurate and consistent with the actual validity status of the certificate at the time the response is generated.
<i>cert/certificates</i>	Required Conditional	<i>Array of String</i>	Contains one or more Base64-encoded X.509v3 certificates from the certificate chain. If the <i>certificates</i> parameter is "chain", the entire certificate chain shall be returned with the end entity certificate at the beginning of the array. If the <i>certificates</i> parameter is "single", only the end entity certificate shall be returned. If the <i>certificates</i> parameter is "none", this parameter shall not be returned.
<i>cert/issuerDN</i>	Required Conditional	<i>String</i>	The Issuer Subject Distinguished Name from the X.509v3 end entity certificate in printable string format, UTF-8-encoded according to RFC 2253. This parameter shall be returned when <i>certInfo</i> is "true".
<i>cert/serialNumber</i>	Required Conditional	<i>String</i>	The Serial Number from the X.509v3 certificate in hex encoded format. This parameter shall be returned when <i>certInfo</i> is "true".
<i>cert/subjectDN</i>	Required Conditional	<i>String</i>	The Distinguished Name from the X.509v3 certificate in printable string format, UTF-8-encoded according to RFC 2253. This parameter shall be returned when <i>certInfo</i> is "true".
<i>cert/validFrom</i>	Required Conditional	<i>String</i>	The validity start date from the X.509v3 certificate in printable string format, encoded as GeneralizedTime format (RFC 2459) (e.g. "YYYYMMDDHHMMSSZ"). This parameter shall be returned when <i>certInfo</i> is "true".
<i>cert/validTo</i>	Required Conditional	<i>String</i>	The validity end date from the X.509v3 certificate in printable string format, encoded as GeneralizedTime format (RFC 2459) (e.g. "YYYYMMDDHHMMSSZ"). This parameter shall be returned when <i>certInfo</i> is "true".
<i>authMode</i>	Required Conditional	<i>String</i> implicit explicit oauth2code oauth2token	Specifies one of the authorization modes: <ul style="list-style-type: none">• "implicit": the authorization process is managed by the Remote Service autonomously.• "explicit": the signature application shall collect up to two levels of security elements.• "oauth2code": the authorization process is managed by the Remote Service using an OAuth 2.0 mechanism based on authorization code as described in Section 1.3.1 of RFC 6749.• "oauth2token": the authorization process is managed by the Remote Service using an OAuth 2.0 mechanism based on implicit grant as described in Section 1.3.2 of RFC 6749.
<i>SCAL</i>	Optional	<i>String</i> 1 2	Specifies the Sole Control Assurance Level required by the credential, as defined in CEN EN 419 241-1: <ul style="list-style-type: none">• "1": at least a basic authorization is required (SCAL1). This level does neither require to invoke any of the credentials authorization methods nor to pass the Signature Activation Data (SAD) to the signatures/signHash method.• "2": at least a two-factor authorization is required (SCAL2). This level requires the Remote Service to generate the Signature Activation Data (SAD).



			This parameter is optional and the default value is "1".
<i>PIN/presence</i>	Required Conditional	<i>String</i> true false optional	Specifies if a text-based PIN is required or not, or optional. This parameter shall be present only when <i>authMode</i> is "explicit".
<i>PIN/format</i>	Required Conditional	<i>String</i> A N	Specifies the format of the PIN: <ul style="list-style-type: none">• "A": the PIN contains alphanumeric text.• "N": the PIN contains numeric text. This parameter shall be present only when <i>authMode</i> is "explicit" and <i>PIN/presence</i> is not "false". The size of the PIN is not specified to improve its security.
<i>PIN/label</i>	Optional Conditional	<i>String</i>	Specifies an optional label for the data field used to collect the PIN in the user interface, in the language specified in the <i>lang</i> parameter. This parameter can be present only when <i>authMode</i> is "explicit" and <i>PIN/presence</i> is not "false".
<i>PIN/description</i>	Optional Conditional	<i>String</i>	It optionally specifies a free form description of the PIN in the language specified in the <i>lang</i> parameter. This parameter can be present only when <i>authMode</i> is "explicit" and <i>PIN/presence</i> is not "false". The maximum size of the string shall be 255 characters.
<i>OTP/presence</i>	Required Conditional	<i>String</i> true false optional	Specifies if a text-based OTP is required or not, or optional. This parameter shall be present only when <i>authMode</i> is "explicit".
<i>OTP/type</i>	Required Conditional	<i>String</i> offline online	Specifies the type of the OTP: <ul style="list-style-type: none">• "offline": The OTP is generated offline by a dedicated device and does not require the client to invoke the credentials/sendOTP method.• "online": The OTP is generated online by the Remote Service when the client invokes the credentials/sendOTP method. This parameter shall be present only when <i>authMode</i> is "explicit" and <i>OTP/presence</i> is not "false".
<i>OTP/format</i>	Required Conditional	<i>String</i> A N	Specifies the data format of the OTP: <ul style="list-style-type: none">• "A": the OTP contains alphanumeric text.• "N": the OTP contains numeric text. This parameter shall be present only when <i>authMode</i> is "explicit" and <i>OTP/presence</i> is not "false".
<i>OTP/label</i>	Optional Conditional	<i>String</i>	Specifies an optional label for the data field used to collect the OTP in the user interface, in the language specified in the <i>lang</i> parameter. This parameter can be present only when <i>authMode</i> is "explicit" and <i>OTP/presence</i> is not "false".
<i>OTP/description</i>	Optional Conditional	<i>String</i>	Optionally specifies a free form description of the OTP mechanism in the language specified in the <i>lang</i> parameter. This parameter can be present only when <i>authMode</i> is "explicit" and <i>OTP/presence</i> is not "false". The maximum size of the string shall be 255 characters.
<i>OTP/ID</i>	Required Conditional	<i>String</i>	Specifies the identifier of the OTP device or application. This parameter shall be present only when <i>authMode</i> is "explicit" and <i>OTP/presence</i> is not "false".
<i>OTP/provider</i>	Optional Conditional	<i>String</i>	Optionally specifies the provider of the OTP device or application. This parameter can be present only when <i>authMode</i> is "explicit" and <i>OTP/presence</i> is not "false".
<i>multisign</i>	Required Conditional	<i>Boolean</i>	Specifies if the credential supports multiple signatures to be created with a single authorization request (e.g. using the transaction signature methods or submitting multiple hash values to the credentials/signHash method).
<i>lang</i>	Optional	<i>String</i>	The language used in the responses, specified according to RFC 3066.



Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
Missing or not String "credentialID" parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter credentialID
Invalid "credentialID" parameter	400 (bad request)	invalid_request	Invalid parameter credentialID
Invalid "certificates" parameter	400 (bad request)	invalid_request	Invalid parameter certificates

Sample Request

```
POST https://service.domain.org/csc/v0/credentials/info
Authorization: Bearer 4/CKN69L8gdSYp5_pwh3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json
{
  "credentialID": "GX0112348",
  "certificates": "chain",
  "certInfo": true,
  "authInfo": true
}
```

Sample Response

```
HTTP/1.1 200 OK
{
  "key":
  {
    "status": "active",
    "algo":
    [
      "1.2.840.113549.1.1.1",
      "0.4.0.127.0.7.1.1.4.1.3"
    ],
    "len": 2048
  },
  "cert":
  {
    "status": "valid",
    "certificates":
    [
      "Base64-encoded X.509 end entity certificate",
      "Base64-encoded X.509 intermediate CA certificate",
      "Base64-encoded X.509 issuer CA certificate"
    ],
    "issuerDN": "The X.500 issuer DN printable string",
    "serialNumber": "5AAC41CD8FA22B953640",
    "subjectDN": "The X.500 subject DN printable string",
    "validFrom": "20160101100000Z",
    "validTo": "20190101095959Z"
  },
  "authMode": "explicit",
  "PIN":
  {
    "presence": true
    "label": "PIN",
  }
}
```



```
    "description": "Please type the signature PIN"
  },
  "OTP":
  {
    "presence": true,
    "type": "offline",
    "ID": "MB01-K741200",
    "provider": "totp",
    "format": "N",
    "label": "Mobile OTP",
    "description": "Please type the 6 digit code you received on your
registered mobile phone"
  },
  "multisign": true,
  "lang": "en-US"
}
```

PRELIMINARY RELEASE

**credentials/authorize**

Description: Authorize the access to the credential for remote signing, according to the authorization mechanisms associated to it. This method returns the Signature Activation Data (SAD) required to authorize the **signatures/signHash** method. PIN and/or OTP values collected from the user shall be present in the request according to the requirements specified by the **credentials/info** method. This method shall be used in case of “explicit” authorization. This method shall also be used in case of “implicit” authorization, to trigger the authorization mechanism managed by the Remote Service. This method shall not be used in case of “oauth2” credential authorization; instead, any of the available OAuth 2.0 authorization mechanisms shall be used.

The *numSignatures* parameter shall indicate the total number of signatures to authorize. In case of multi-signature transaction obtained by invoking the **signatures/signHash** multiple times, the Signature Application should obtain a new SAD by invoking the **credentials/extendTransaction** method before the current SAD expires.

Input: This method allows the following parameters:

Parameter	Presence	Value	Description
<i>credentialID</i>	Required	<i>String</i>	The identifier associated to the credential.
<i>numSignatures</i>	Required	<i>Number</i>	The number of signatures to authorize. Multi-signature transactions can be obtained by using a combination of passing an array of hash values and calling the signatures/signHash method multiple times.
<i>hash</i>	Required Conditional	<i>Array of String</i>	One or more Base64-encoded hash values to be signed. It shall be used if the <i>SCAL</i> parameter returned by credentials/info is “2”.
<i>PIN</i>	Required Conditional	<i>String</i>	The PIN collected from the user. It shall be used only when <i>authMode</i> from credentials/info is “explicit” and <i>PIN/presence</i> is not “false”.
<i>OTP</i>	Required Conditional	<i>String</i>	The OTP collected from the user. It shall be used only when <i>authMode</i> from credentials/info is “explicit” and <i>OTP/presence</i> is not “false”.
<i>description</i>	Optional	<i>String</i>	Contains a free form description of the authorization transaction in the <i>lang</i> language. The maximum size of the string shall be 500 characters. It can be useful when <i>authMode</i> from credentials/info is “implicit” to provide some hints about the occurring transaction.
<i>clientData</i>	Optional	<i>String</i>	Arbitrary data from the Signature Application. It can be used to handle a transaction identifier or other application-specific data.

Output: This method returns the following parameters:

Parameter	Presence	Value	Description
<i>SAD</i>	Required	<i>String</i>	The Signature Activation Data to provide as input to the signatures/signHash method.
<i>expiresIn</i>	Optional	<i>Number</i>	The lifetime in seconds of the service access token. If omitted, the default expiration time is 3600 (1 hour).



Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
Missing or not String "credentialID" parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter credentialID
Invalid "credentialID" parameter	400 (bad request)	invalid_request	Invalid parameter credentialID
Missing or not integer "numSignatures" parameter	400 (bad request)	invalid_request	Missing (or invalid type) integer parameter numSignatures
"numSignatures" < 1	400 (bad request)	invalid_request	Invalid parameter numSignatures
When present, invalid "clientData" format (not string)	400 (bad request)	invalid_request	Invalid parameter clientData
Invalid OTP	400 (bad request)	invalid_otp	The OTP is invalid
Invalid PIN	400 (bad request)	invalid_pin	The PIN is invalid
PIN locked	400 (bad request)	invalid_request	PIN locked
OTP locked	400 (bad request)	invalid_request	OTP locked

Sample Request

```
POST https://service.domain.org/csc/v0/credentials/authorize
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json
{
  "credentialID": "GX0112348",
  "numSignatures": 2,
  "hash":
  [
    "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
    "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFdlST0="
  ],
  "PIN": "12345678",
  "OTP": "738496",
  "clientData": "12345678"
}
```

Sample Response

```
HTTP/1.1 200 OK
{
  "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw"
}
```



credentials/extendTransaction

Description: Extends the validity of a multi-signature transaction authorization by obtaining a new SAD. This method shall be used in case of multi-signature transaction obtained by invoking the **signatures/signHash** multiple times. It can also be used to renew a SAD, before it expires, when signature operations take longer than the amount of time returned in *expiredIn* by the **credentials/authorize** method. Expired SAD cannot be extended. The SAD would also automatically expire when the maximum number of authorized signatures specified in *numSignatures* is reached.

Input: This method allows the following parameters:

Parameter	Presence	Value	Description
<i>credentialID</i>	Required	String	The identifier associated to the credential.
<i>SAD</i>	Required	String	The current Signature Activation Data. This token is returned by the credentials/authorize or by the previous credentials/extendTransaction methods.
<i>clientData</i>	Optional	String	Arbitrary data from the Signature Application. It can be used to handle a transaction identifier or other application-specific data.

Output: This method returns the following parameters:

Parameter	Presence	Value	Description
<i>SAD</i>	Required	String	The new Signature Activation Data required to sign multiple times with a single authorization.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.

Sample Request

POST <https://service.domain.org/csc/v0/credentials/extendTransaction>

Authorization: Bearer 4/CKN69L8gdSYp5_pwh3XlFQZ3ndFhkXf9P2_TiHRG-bA

Content-Type: application/json

```
{
  "credentialID": "GX0112348",
  "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
  "clientData": "12345678"
}
```

Sample Response

HTTP/1.1 200 OK

```
{
  "SAD": "1/UsHDJ98349h9fgh9348hKKHDkHWVkl/8hsAW5usc8_5="
}
```



credentials/sendOTP

Description: Start the online OTP generation mechanism associated with a credential and managed by the Remote Service. This will generate a dynamic one-time password that will be delivered to the user associated with the credential through an agreed communication channel (e.g. SMS, email, etc.).
This method shall only be used with “online” OTP generators. In case of “offline” OTP, the Signature Application should not invoke this method.

Input: This method allows the following parameters:

Parameter	Presence	Value	Description
<i>credentialID</i>	Required	<i>String</i>	The identifier associated to the credential.
<i>clientData</i>	Optional	<i>String</i>	Arbitrary data from the Signature Application. It can be used to handle a transaction identifier or other application-specific data.

Output: This method returns no parameters.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern “Bearer [sessionKey]”	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
Missing or not String “credentialID” parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter credentialID
Invalid “credentialID” parameter	400 (bad request)	invalid_request	Invalid parameter credentialID
OTP locked	400 (bad request)	invalid_request	OTP locked

Sample Request

```
POST https://service.domain.org/csc/v0/credentials/sendOTP
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json
{
  "credentialID": "GX0112348",
  "clientData": "12345678"
}
```

Sample Response

```
HTTP/1.1 200 OK
```



signatures/signHash

Description: Calculate the remote digital signature of one or multiple hash values provided as an input. This method requires providing credential authorization in the form of Signature Activation Data (SAD). The Signature Application shall first obtain the SAD from any of the **credential/authorize** or the **oauth2/authorize** methods, depending on the type of supported authorization mechanisms associated with the credential, and shall pass it as an input to this method. In case of multi-signature transactions, the SAD shall be updated with **credentials/extendTransaction** every time this method is invoked until the maximum number of authorized signatures has been generated.

Input: This method allows the following parameters:

Parameter	Presence	Value	Description
<i>credentialID</i>	Required	<i>String</i>	The identifier associated to the credential.
<i>SAD</i>	Required	<i>String</i>	The Signature Activation Data returned by the Credential Authorization methods.
<i>hash</i>	Required	<i>Array of String</i>	One or more Base64-encoded hash values to be signed.
<i>hashAlgo</i>	Required Conditional	<i>String</i>	Specifies the OID of the algorithm used to calculate the hash value(s), in case it's not implicitly specified by the <i>signAlgo</i> algorithm. Only hashing algorithms as strong or stronger than SHA256 shall be used.
<i>signAlgo</i>	Required	<i>String</i>	Specifies the OID of the algorithm to use for signing. It shall be one of the values allowed by the credential as returned in <i>keyAlgo</i> by the credentials/info method.
<i>signAlgoParams</i>	Required Conditional	<i>String</i>	Specifies the Base64-encoded or DER-encoded ASN.1 signature parameters, if required by the signature algorithm. Some algorithms like RSA-PSS [RFC 3447] may require additional parameters.
<i>clientData</i>	Optional	<i>String</i>	Arbitrary data from the Signature Application. It can be used to handle a transaction identifier or other application-specific data.

Output: This method returns the following parameters:

Parameter	Presence	Value	Description
<i>signatures</i>	Required	<i>Array of String</i>	One or more Base64-encoded signed hash. In case of multiple signatures, the signed hashes shall be returned in the same order of the corresponding hashes provided as an input parameter.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.
Missing or not String "SAD" parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter SAD
Invalid "SAD" parameter	400 (bad request)	invalid_request	Invalid parameter SAD
Missing or not String "credentialID" parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter credentialID
Invalid "credentialID" parameter	400 (bad request)	invalid_request	Invalid parameter credentialID



Missing or not Array "hash" parameter	400 (bad request)	invalid_request	Missing (or invalid type) array parameter hash
Empty hash parameter	400 (bad request)	invalid_request	Empty hash array
Invalid Base64 hash element	400 (bad request)	invalid_request	Invalid Base64 hash string parameter
Missing or not String "signAlgo" parameter	400 (bad request)	invalid_request	Missing (or invalid type) string parameter signAlgo
Missing or not String "hashAlgo" parameter when "signAlgo" is equal to "1.2.840.113549.1.1.1"	400 (bad request)	invalid_request	Missing (or invalid type) string parameter hashAlgo
Invalid "hashAlgo" parameter	400 (bad request)	invalid_request	Invalid parameter hashAlgo
Invalid "signAlgo" parameter	400 (bad request)	invalid_request	Invalid parameter signAlgo
When present, invalid "clientData" format (not string)	400 (bad request)	invalid_request	Invalid parameter clientData
Invalid "hash" length	400 (bad request)	invalid_request	Invalid digest value length
The OTP used to generate the "SAD" is invalid	400 (bad request)	invalid_otp	The OTP is invalid
Expired credential	400 (bad request)	invalid_request	Signing certificate 'O=[organization],CN=[common_name]' is expired.

Sample Request

```
POST https://service.domain.org/csc/v0/signatures/signHash
Authorization: Bearer 4/CKN69L8gdSYp5_pwh3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json
{
  "credentialID": "GX0112348",
  "SAD": "_TiHRG-bAH3XlFQZ3ndFhkXf9P24/CKN69L8gdSYp5_pw",
  "hash":
  [
    "sTOgwOm+474gFj0q0xliSNspKqbcse4IeiqlDg/HWuI=",
    "c1RPZ3dPbSs0NzRnRmowcTB4MWlTTnNwS3FiY3NlNEllaXFsRGcvSFd1ST0="
  ],
  "hashAlgo": "2.16.840.1.101.3.4.2.1",
  "signAlgo": "1.2.840.113549.1.1.1",
  "clientData": "12345678"
}
```

Sample Response

```
HTTP/1.1 200 OK
{
  "signatures":
  [
    "KedJuTob5gtvYx9qM3k3gm7kbLBwVbEQRl26S2tmXjqNND7MRGtoew==",
    "Idhef7xzgtvYx9qM3k3gm7kbLBwVbE98239S2tm8hUh85KKsfowel=="
  ]
}
```



signatures/timestamp

Description: Generate a time-stamp token for the input hash value. The time-stamp token can be generated directly by the RSSP or by a Time Stamping Authority connected to it. The reason to implement this method instead of providing time-stamp services through RFC 3161 protocols directly is to facilitate the creation of long-term validation digital signatures and to support billing operations. In both cases, the RSSP provider can offer pre-configured time-stamp services instead of requiring the Signature Application to obtain time-stamp services from a different provider.

Input: This method allows the following parameters:

Parameter	Presence	Value	Description
<i>hash</i>	Required	<i>String</i>	The Base64-encoded hash value to be time stamped. The Remote Service uses the input value to encode the MessageImprint.hashedMessage value.
<i>hashAlgo</i>	Required	<i>String</i>	Specifies the OID of the algorithm used to calculate the hash value. The Remote Service uses the input value to encode the MessageImprint.hashAlgorithm value.
<i>nonce</i>	Optional	<i>String</i>	Specifies a large random number with a high probability that it is generated by the Signature Application only once.
<i>clientData</i>	Optional	<i>String</i>	Arbitrary data from the Signature Application. It can be used to handle a transaction identifier or other application-specific data.

Output: This method returns the following parameters:

Parameter	Presence	Value	Description
<i>timestamp</i>	Required	<i>String</i>	The Base64-encoded time-stamp token as defined in RFC 3161 as updated by RFC 5816. If the <i>nonce</i> parameter is included in the request then it shall also be included in the time-stamp token, otherwise the response shall be rejected.

Error Case	Status Code	Error	Error Description
The authorization header does not match the pattern "Bearer [sessionKey]"	400 (bad request)	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.

Sample Request

```
POST https://service.domain.org/csc/v0/signatures/timestamp
Authorization: Bearer 4/CKN69L8gdSYp5_pwH3XlFQZ3ndFhkXf9P2_TiHRG-bA
Content-Type: application/json
{
  "hash": "sTOgwOm+474gFj0q0x1iSNspKqbcse4IeiqlDg/HWuI=",
  "hashAlgo": "2.16.840.1.101.3.4.2.1",
  "clientData": "12345678"
}
```

Sample Response



```
HTTP/1.1 200 OK
{
  "timestamp":
  "MGwCAQEGCSsGAQQB7U8CATAxMA0GCWCGSAFlAwQCAQUABCCrCqnrjH0VxXyQQlfnFJRxljjrviTs
  7/GjKghr2AmluQIIVs5D8OUB4p4YDzIwMTQxMTE5MTEzMjM5WjADAgEBAgkAnWn2SSIWlXk="
}
```

PRELIMINARY RELEASE

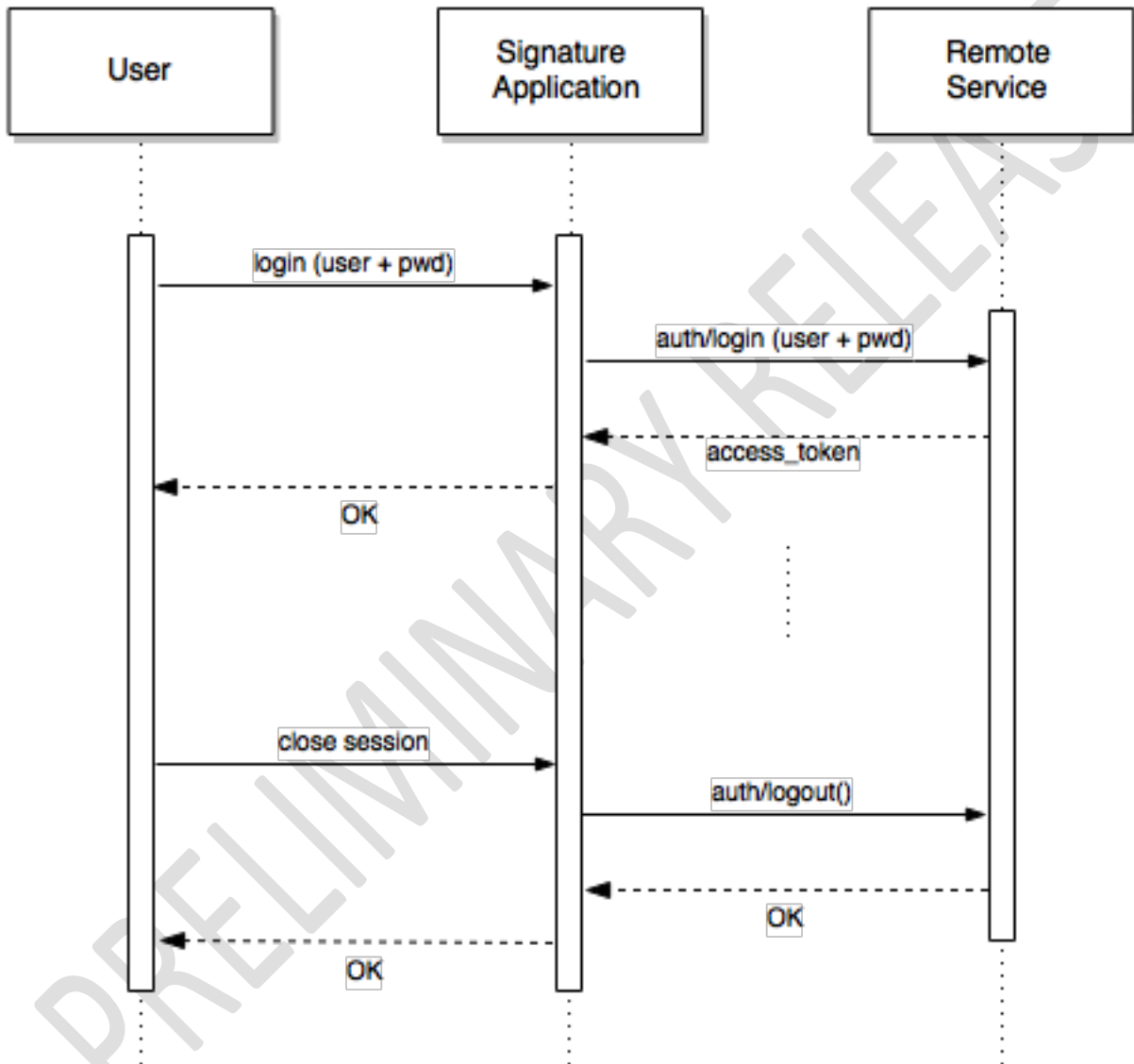
13 Interaction among elements and components

The building blocks of a Remote Signature solution interact with the API methods described in this specification according to the relations described in the following sections.

13.1 Acquire the context of a RSSP

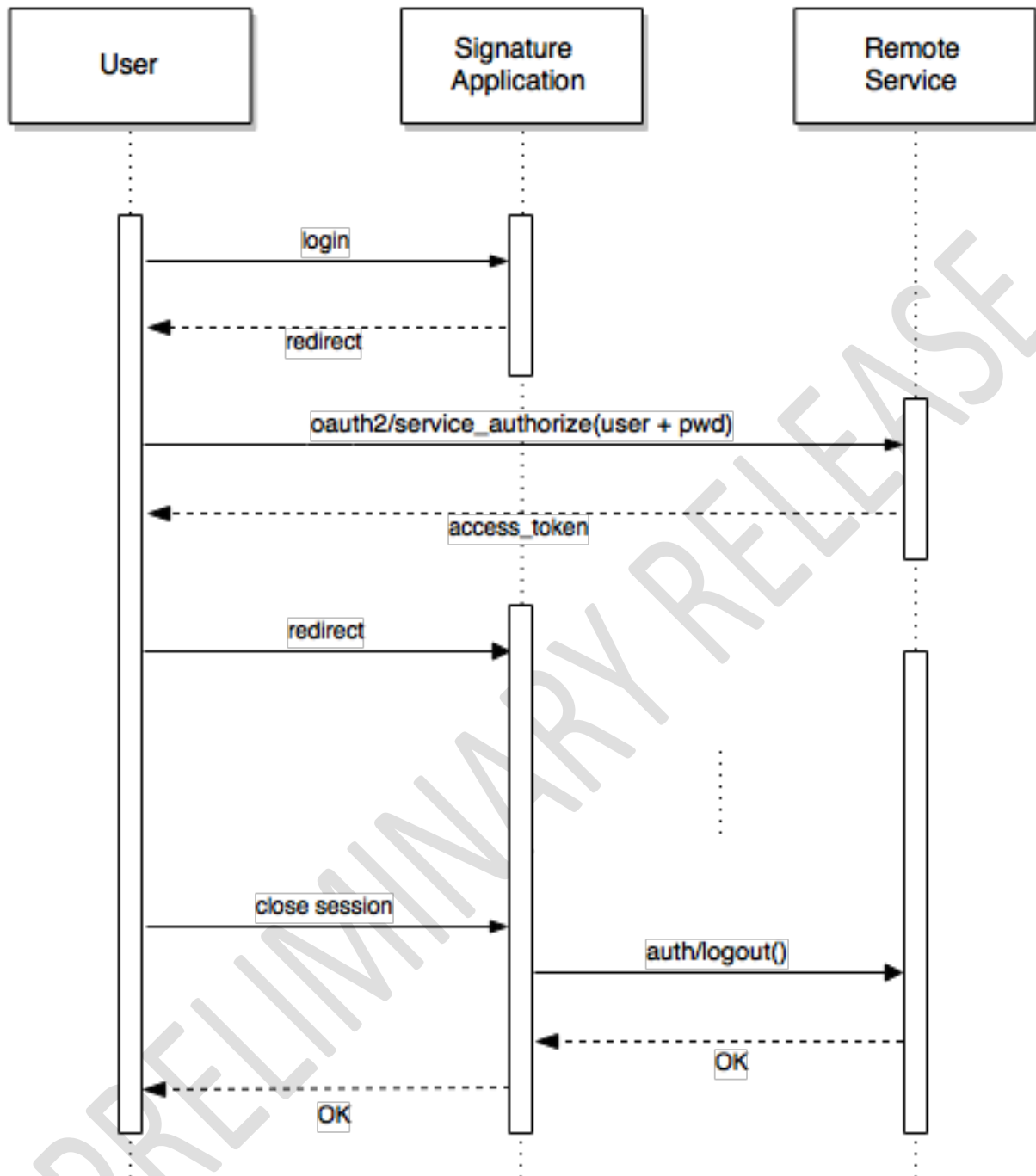
<TBD – Describe the use of the info method>

13.2 RSSP service authorization using a username and password





13.3 RSSP service authorization using OAuth2 with Implicit Grant flow



13.4 RSSP service authorization using OAuth2 with Authorization Code flow

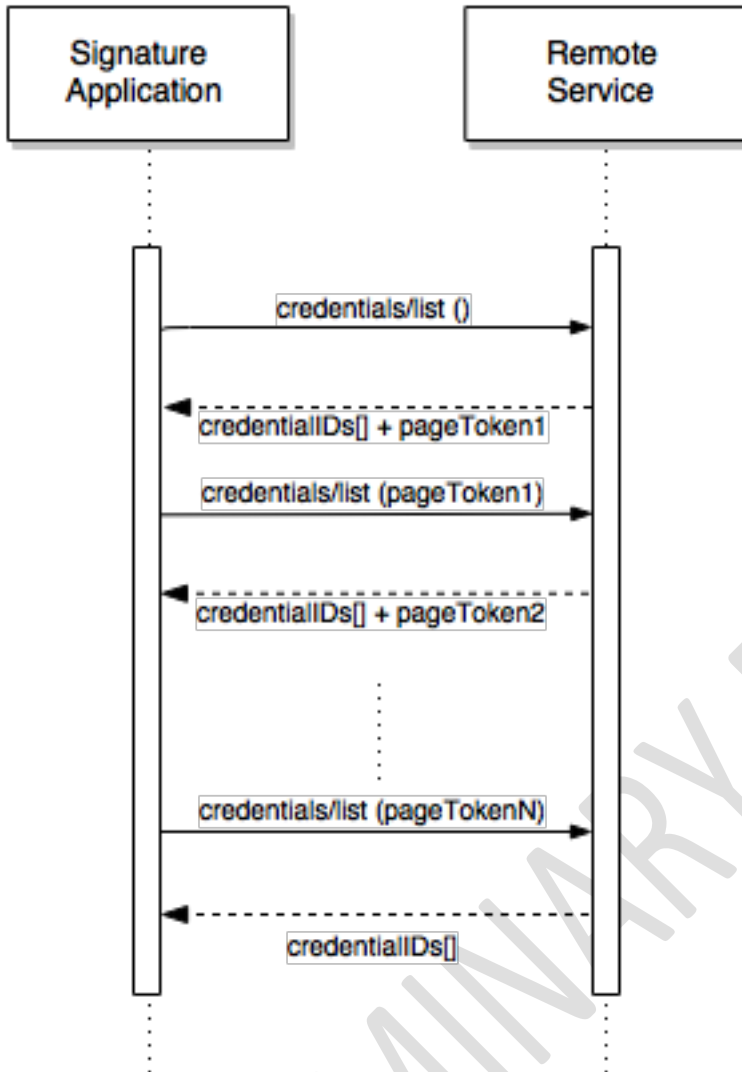
<TBD>

13.5 RSSP service authorization using OAuth2 with Client Credentials flow

<TBD>



13.6 Get the list of credentials available for a group or community of users

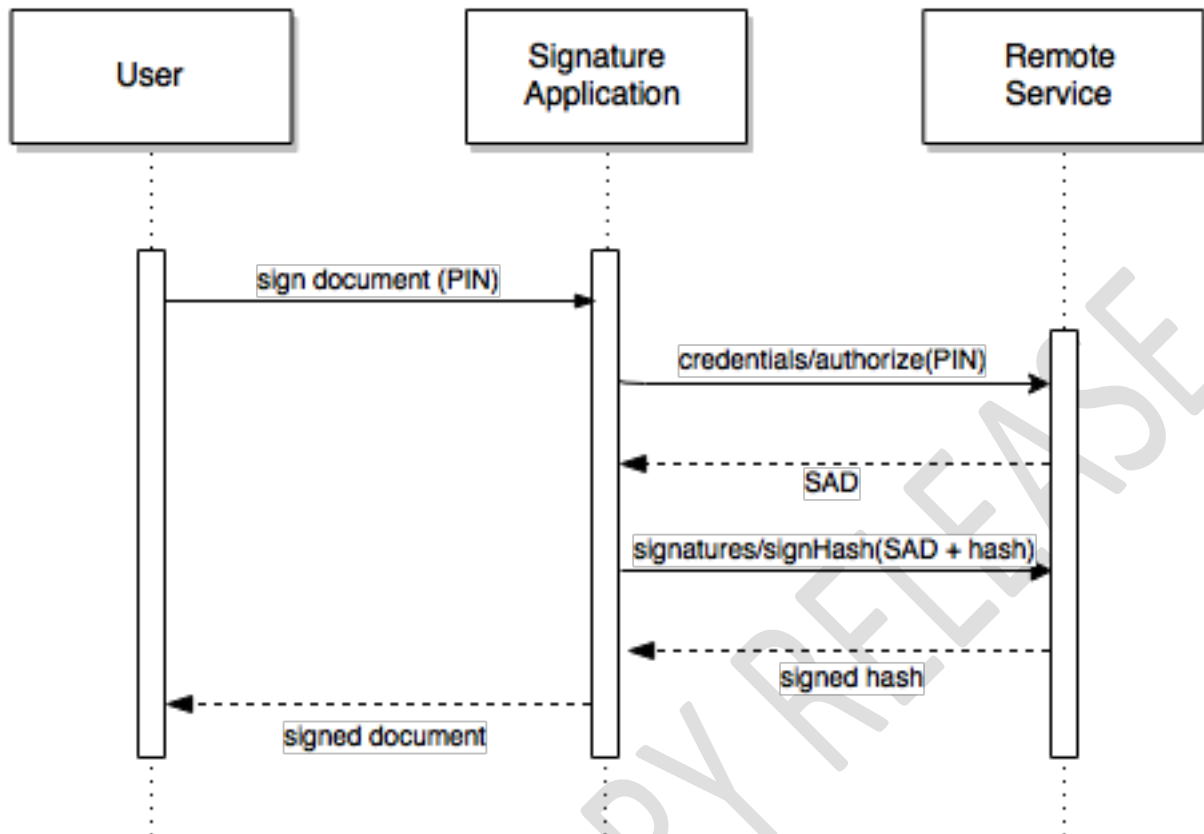


13.7 Create a remote signature with a credential protected by an implicit authorization

<TBD>

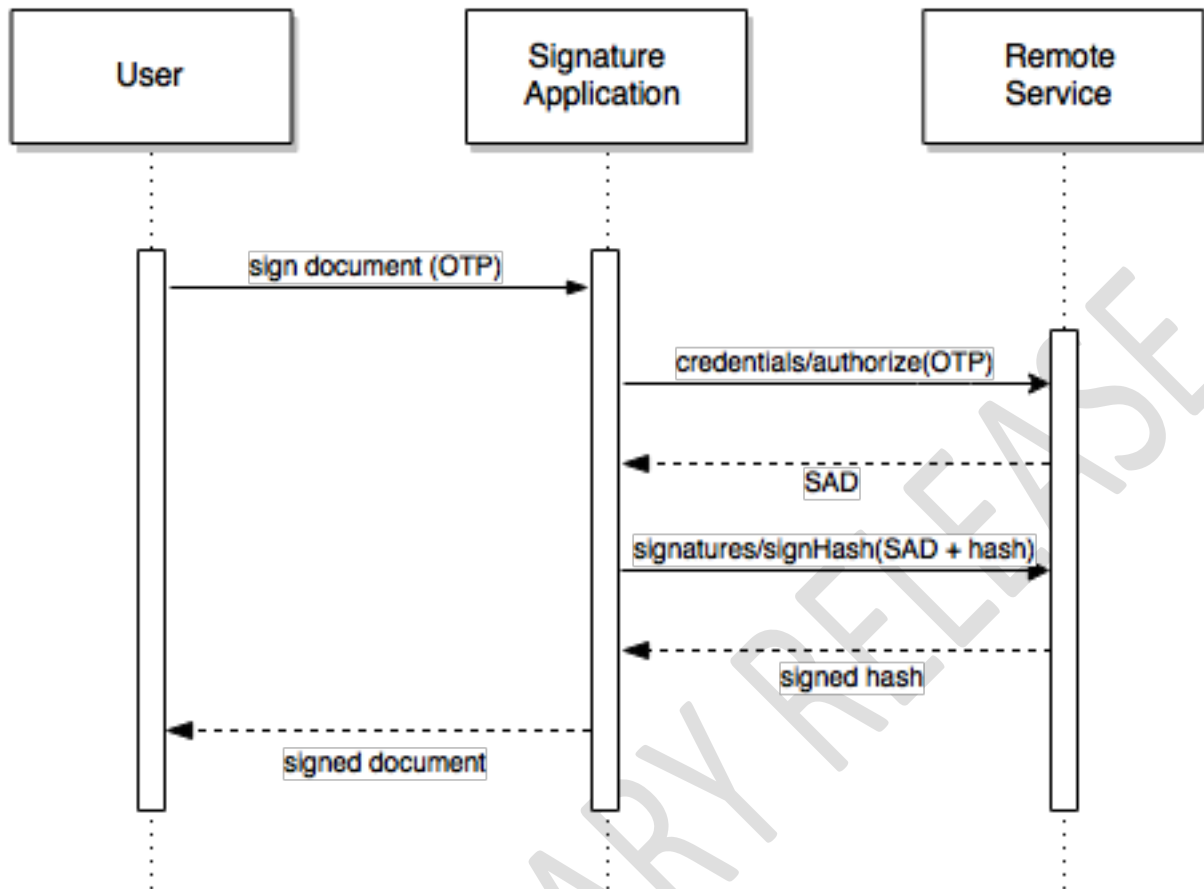


13.8 Create a remote signature with a credential protected by a PIN



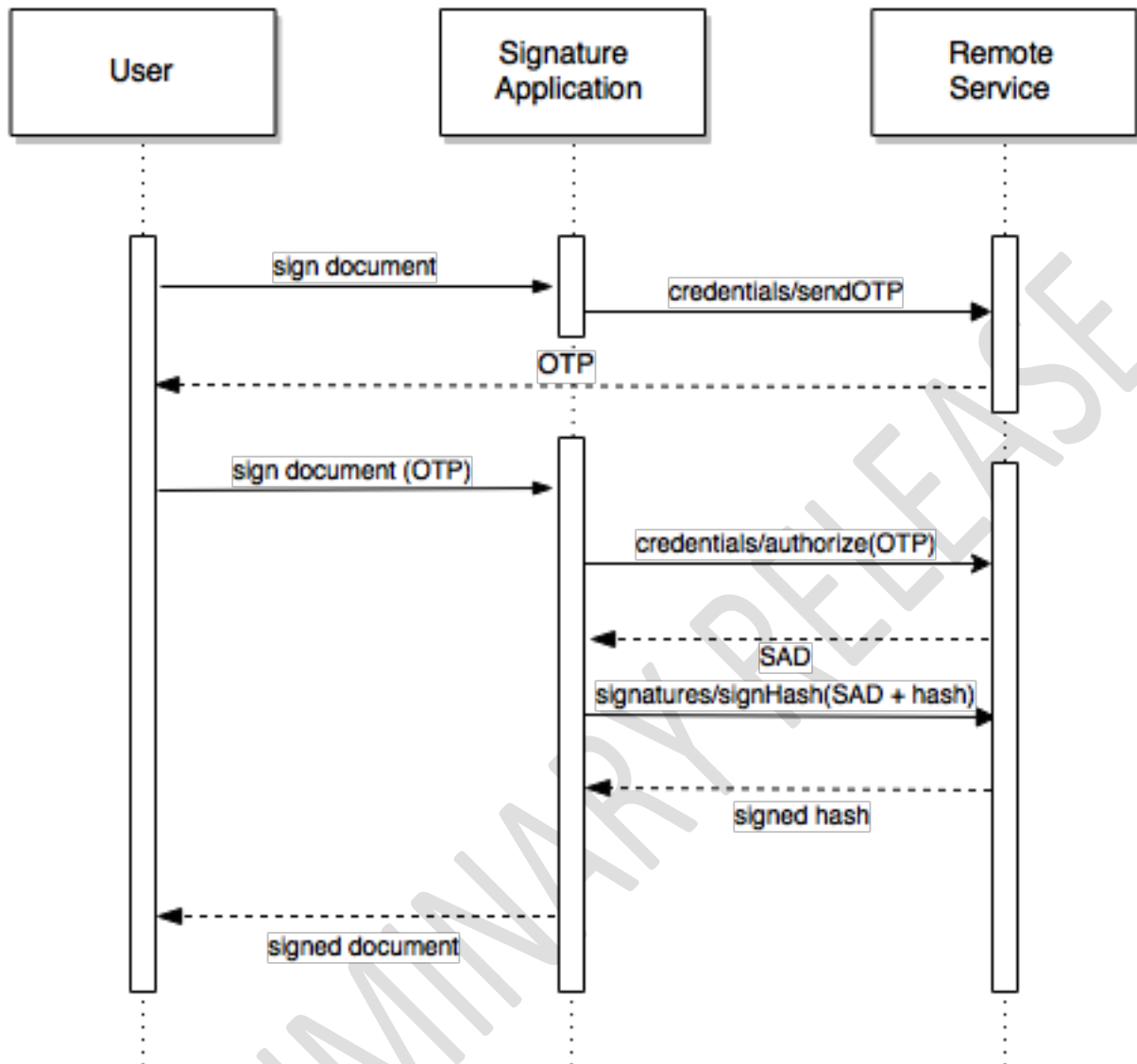


13.9 Create a remote signature with a credential protected by an “offline” OTP



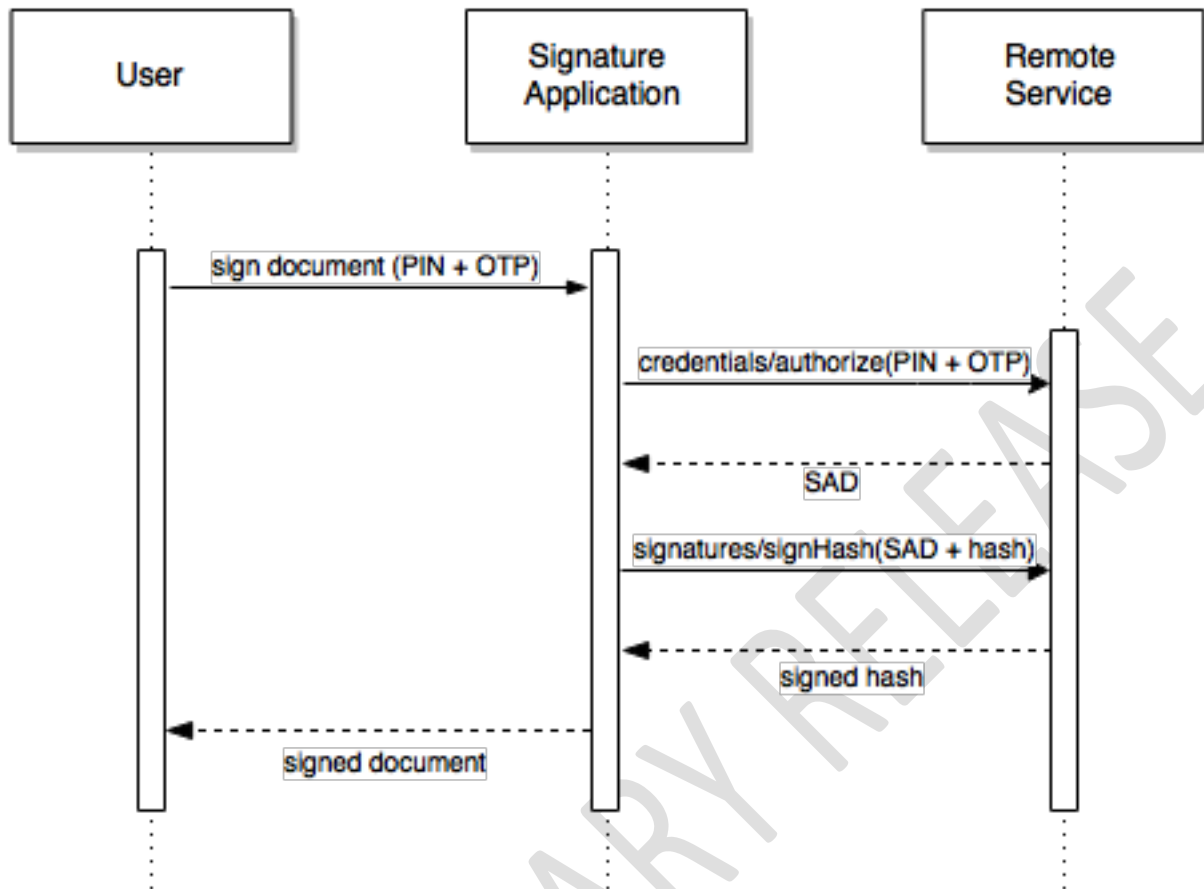


13.10 Create a remote signature with a credential protected by an “online” OTP



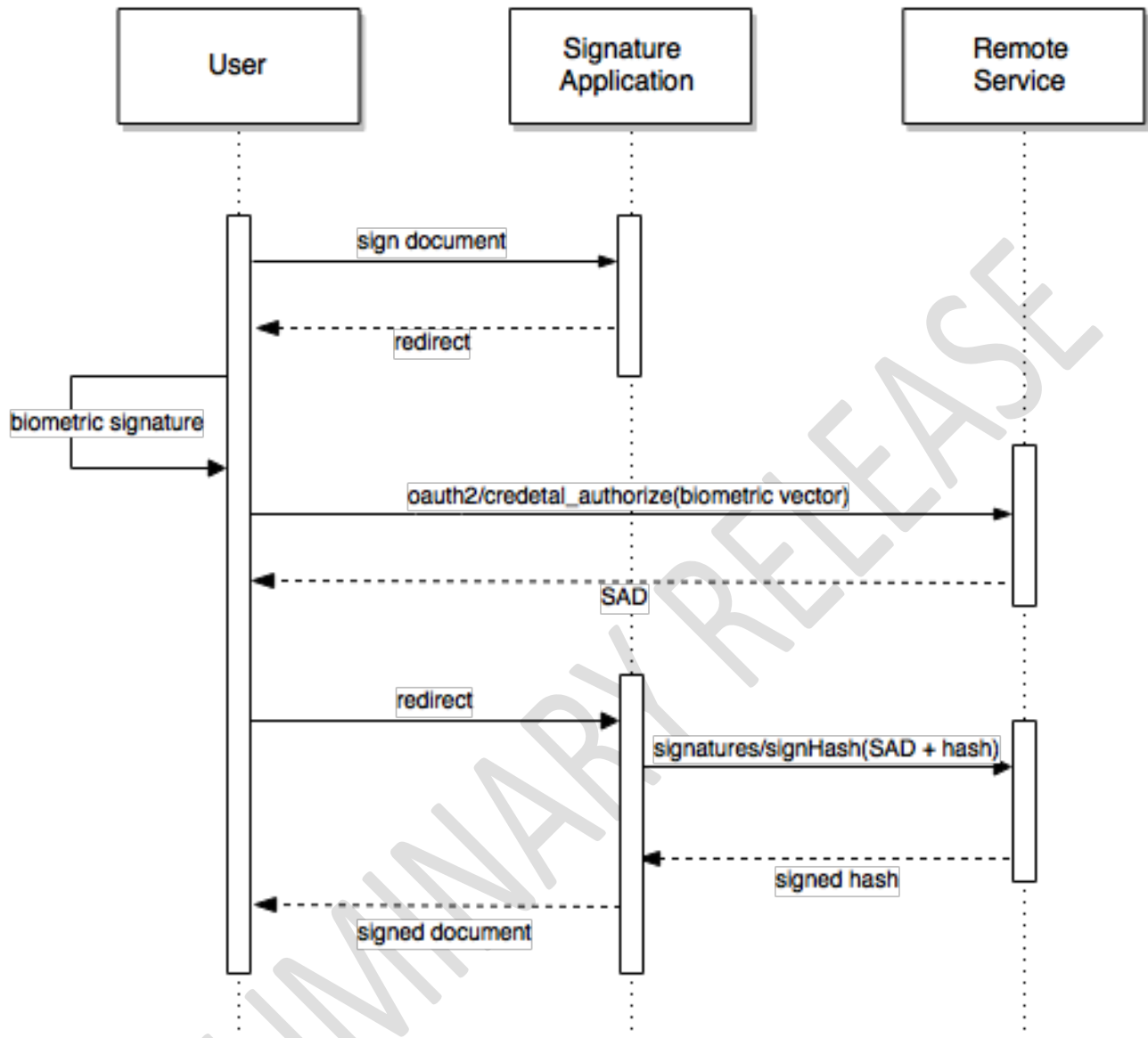


13.11 Create a remote signature with a credential protected by PIN and OTP



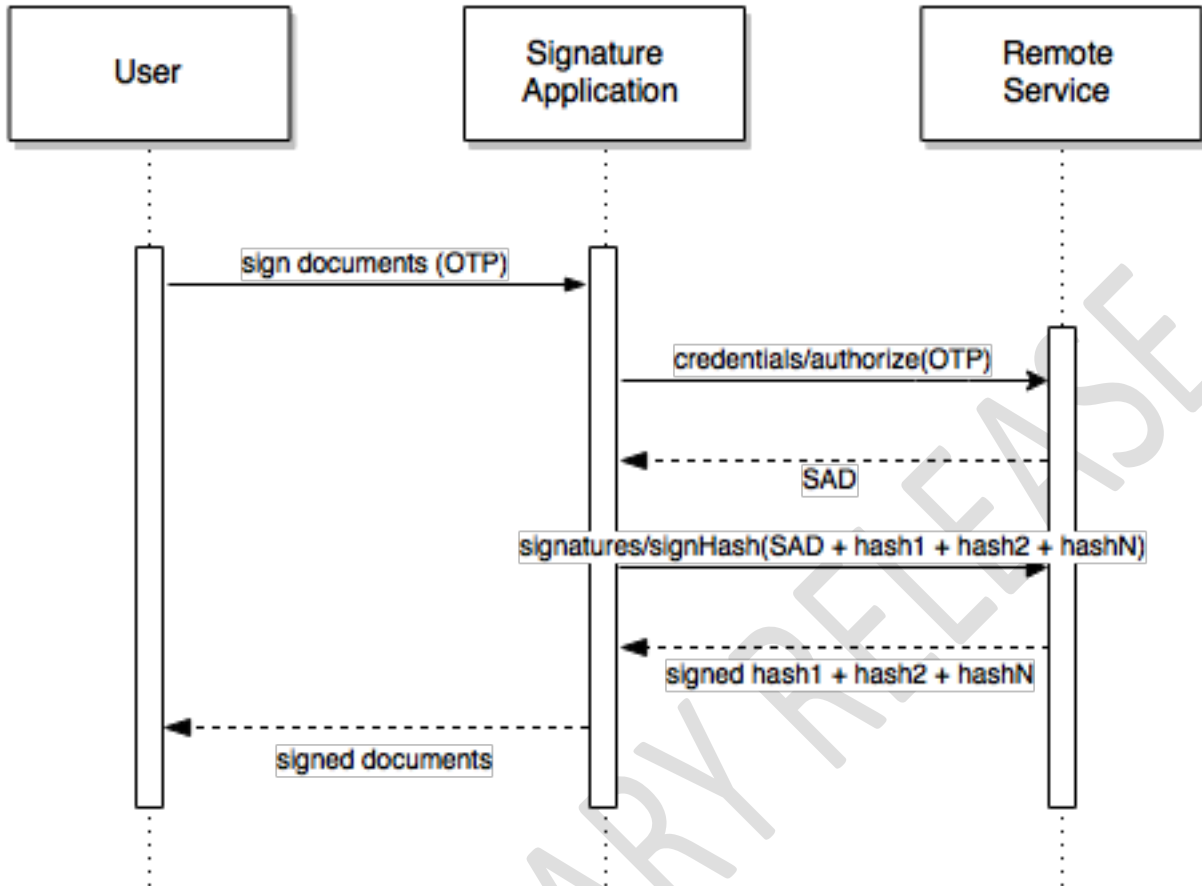


13.12 Create a remote signature with a credential protected by a biometric factor



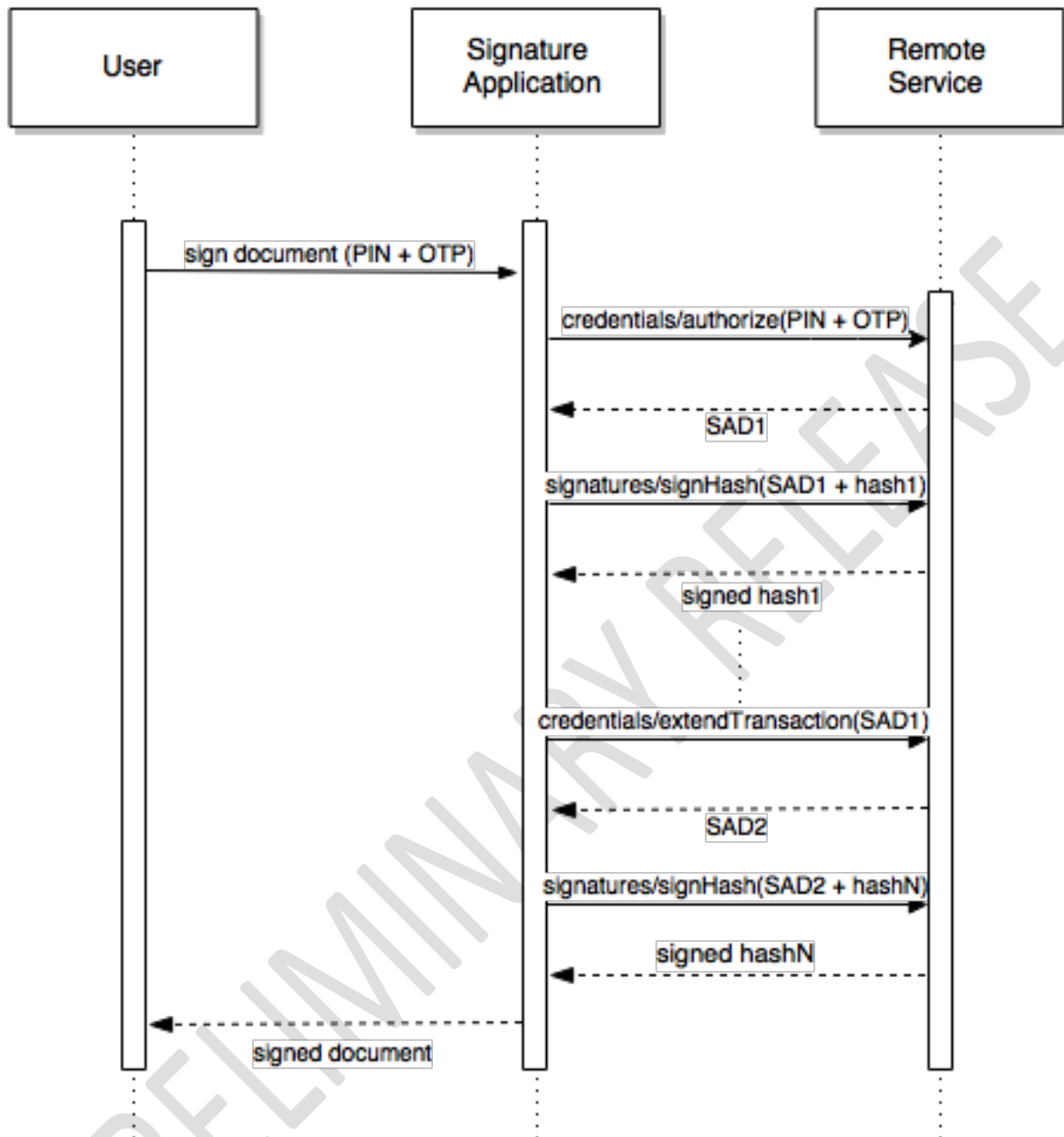


13.13 Create multiple remote signatures from a list of hash values



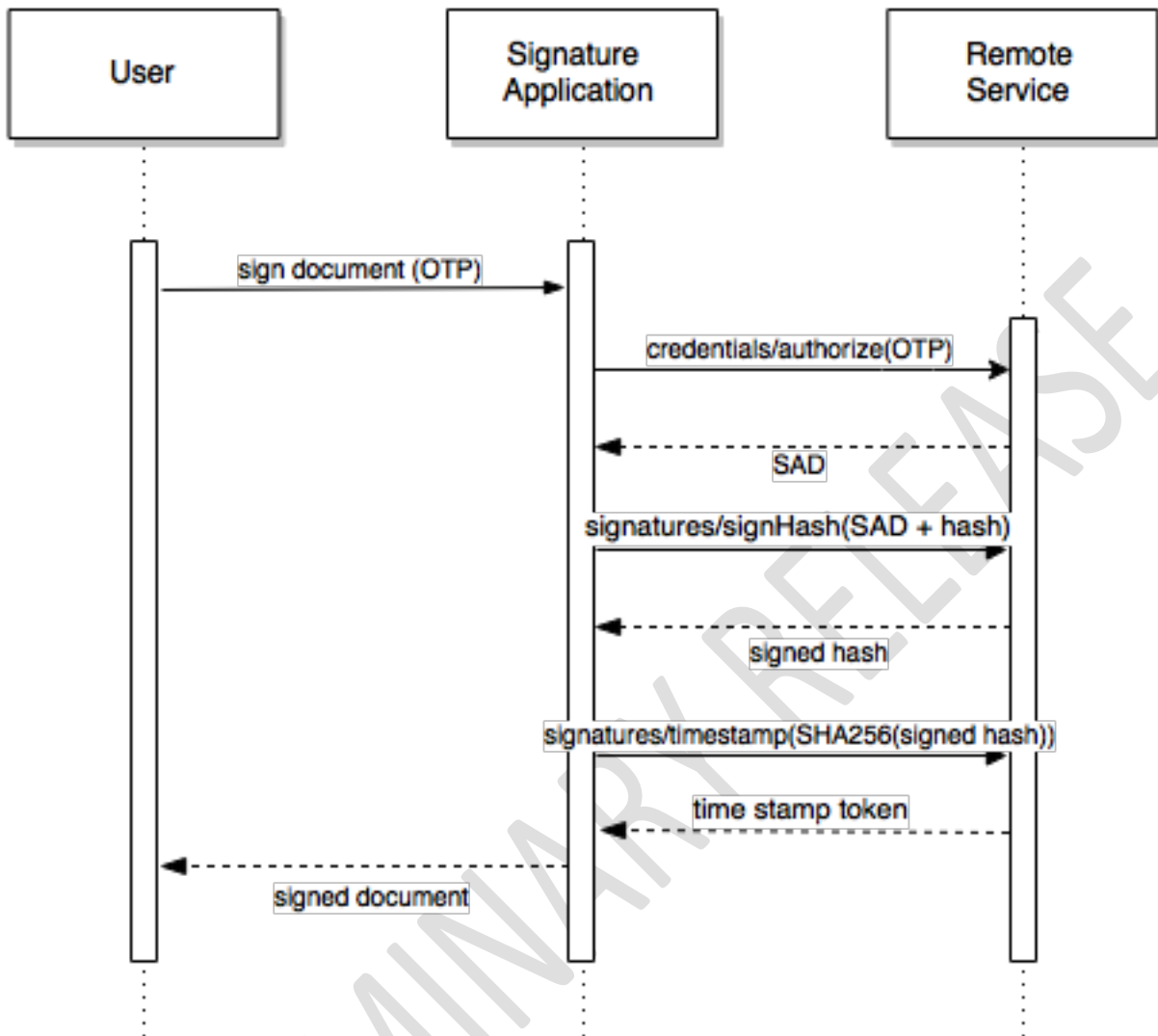


13.14 Create a remote multi-signatures transaction





13.15 Create a remote signature with long-term validity profile





PRELIMINARY RELEASE