Liberty ID-WSF Advanced Client Technologies Overview

Version: 1.0

Editors:
Conor P. Cahill, Intel Corporation

Contributors:
Shelagh Callahan, Intel Corporation
Jane Dashevsky, Intel Corporation
Tapio Kaukonen, Nokia
Sampo Kellomäki, Symlabs, Inc.
Hubert Le Van Gong, Sun
Andrew Lindsay-Stewart, Vodafone
Paul Madsen, NTT
Paul Miller, Gemplus
Hiroyoshi Takiguchi, NTT
Pierre Vannel, Gemplus
Sean Walker, Axalto

Abstract:

This specification defines mechanisms by which smart clients can operate in disconnected mode while accessing and providing web services.

Filename: liberty-idwsf-adv-client-v1.0.pdf

Notice

Liberty Alliance Project
Licensing Administrator
c/o IEEE-ISTO
445 Hoes Lane
Piscataway, NJ 08855-1331, USA
info@projectliberty.org

Contents

1. Introduction

This document provides an overview of the Liberty ID-WSF Advanced Client Technologies. These technologies encompass a suite of advanced functionality in the areas of SSO, federation, service hosting, reporting and provisioning.

1.1. Notation and Conventions

This specification uses schema documents conforming to W3C XML Schema (see [Schema1-2]) and normative text to describe the syntax and semantics of XML-encoded messages.

The key words "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

These keywords are thus capitalized when used to unambiguously specify requirements over protocol and application features and behavior that affect the interoperability and security of implementations. When these words are not capitalized, they are meant in their natural-language sense.

1.1.1. XML Namespaces

The following XML namespaces are referred to in this document:

- The prefix *prov:* stands for the Liberty ID-WSF Provisioning Service namespace [LibertyPROV]:

  urn:liberty:prov:2007-09

- The prefix *pmm:* stands for the Liberty ID-WSF Provisioned Module Manager Service namespace [LibertyPMM]:

  urn:liberty:pmm:2007-09

- The prefix *shps:* stands for the Liberty ID-WSF Service Hosting/Proxying Service namespace [LibertySHPS]:

  urn:liberty:shps:2007-09

- The prefix *idp:* stands for the Liberty ID-WSF IdP Service namespace [LibertyIdP]:

  urn:liberty:idp:2007-09

- The prefix *saml2:* stands for the SAMLv2 assertion namespace [SAMLCore2]:

  urn:oasis:names:tc:SAML:2.0:assertion

- The prefix *samlp2:* stands for the SAMLv2 protocol namespace [SAMLCore2]:

  urn:oasis:names:tc:SAML:2.0:protocol

- The prefix *xs:* stands for the W3C XML schema namespace [Schema1-2]:

  http://www.w3.org/2001/XMLSchema

- The prefix *xsi:* stands for the W3C XML schema instance namespace:

  http://www.w3.org/2001/XMLSchema-instance

2. Basic Concepts

2.1. Actors

There are a number of potential actors that may participate in the various transactions we envision – some familiar ID-WSF actors that we all know and love and some new actors created specifically here to provide some new, unanticipated functionality. These actors include (in no special order):

- Identity Provider (IdP) - the standard Liberty IdP. This can be an independent third party or it can be the TM itself (for self asserted identities).
- Provisioned Module (PM) - a software module that has been provisioned to a device using the protocols documented in this specification. The specific functionality contained within the PM is not defined or restricted by this specification.
- Trusted Module (TM) - a PM which is an extension of the IdP and implements some of the SSO and federation capabilities discussed in this document. The word *trusted* appears in the name to indicate that the client has sufficient protections to enable a third party to trust its participation in some of the more sensitive transactions (although the existence of, type of, and proof of these protections is not in scope of this specification and there are some cases where no such protections are necessary).
- Trusted Environment - an area on the advanced client where TMs are provisioned providing some level of tamper resistance. In some cases, the entire Advanced Client is a trusted environment.
- Provisioning Service (ProvS) - an entity that supports the provisioning of PMs to the client platform.
- Provisioned Module Manager (PMM) - an entity that manages modules (Trusted and non-trusted) and is responsible for performing local provisioning of these modules to the advanced client (at the direction of the provisioning party).
- Client Service Instance (CSI) - a web service instance hosted on an intelligent client.
- Advanced Client (Advanced Client) - a general term for an entity which has some set of one or more PMMs, PMs, TMs, and/or CSIs.
- Service Hosting/Proxying Service (SHPS) - a network visible agent for the CSI which may participate in some transactions in order to facilitate increased availability of locally hosted services or privacy.
- Registration Service (RegS) - an application which typically initiates a provisioning process through the ProvS. The RegS is frequently associated with an IdP.
- Service Provider (SP) - the consumer of browser based SSO transactions. This is a standard SAML 2.0 actor.
- Web Service Provider (WSP) - the standard Liberty ID-WSF provider of web services.
- Web Service Consumer (WSC) - the standard Liberty ID-WSF consumer of web services.
- Internet Browser - just your run-of-the-mill browser (e.g., Mozilla, Internet Explorer, etc.) which is a typical participant in SP SSO transactions.

The figure below shows the relationships between the various actors described above.

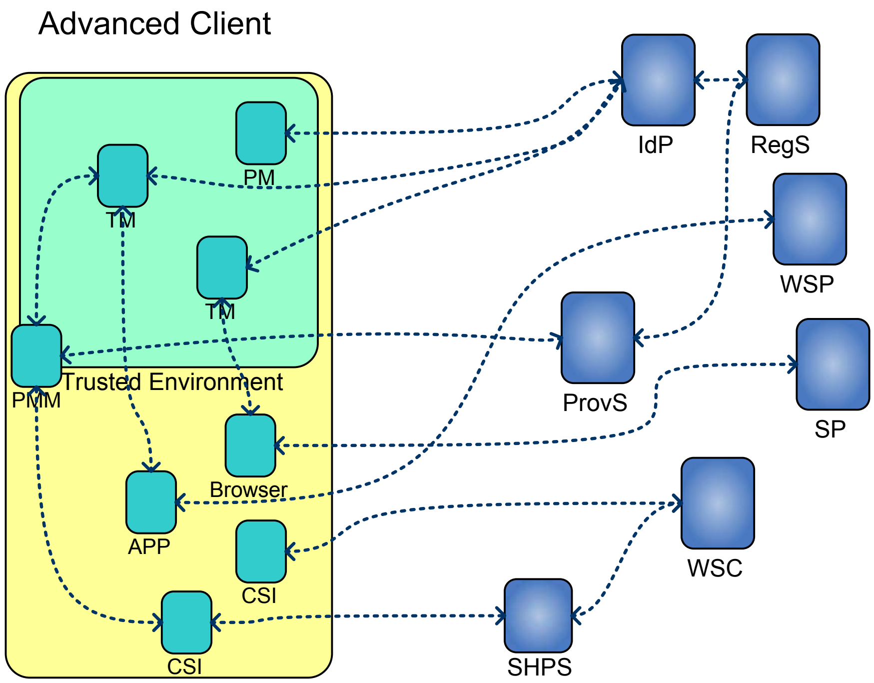

Figure 1. Actors

2.2. Transactions

This specification documents the protocols and paradigms necessary to support the following types of transactions:

- **An extension of the IdP** - the TM may serve as an extension of the IdP performing delegated transactions outlined below. The reasons for using the TM in such a manner are varied, but include: distributed processing, privacy and IdP availability.

  - **Federation** transactions. These transactions involve the establishment of a new identity relationship with an SP/WSP on behalf of a principal. These transactions can include delegated (where the TM is acting as an agent of the IdP), facilitated (where the TM facilitates federation operations – e.g. SAML 2.0 Enhanced Client/Proxy), and self asserted (where the TM **is** the IdP for the identities being federated) federations.

    Delegated operations allow the TM to operate independently of the IdP (so that a federation can take place when the IdP is not visible to the TM). However, the TM must obtain potential federation handles from the IdP prior to disconnecting in order to ensure that both the IdP and TM don't create different federation handles for the principal while the TM is disconnected.

    Self federations allow the TM to create and federate locally generated identities with SPs for use in transactions that don't require a third party asserted identity.

  - **SSO** transactions with both SPs and WSPs. Like Federation transactions, SSO transactions can be delegated, facilitated, and self-asserted. The delegated model, also creates potential privacy preserving separation of assertion use from the IdP.

    As part of an SSO transaction the TM may need to authenticate the user to the IdP. This would typically be accomplished using one of the following methods:

    - collect credentials from the principal and relay them to the IdP for validation.
    - assert internally stored credentials to the IdP as an authentication for the principal (similar to what a SIM card does within a mobile phone).
    - locally authenticate the principal through various means (PIN, PW, biometrics, etc.) and then using an internal credential, assert the completion of that authentication to the IdP.

- **Web Service Instance** - the client hosts one or CSIs. For privacy and/or availability reasons, the CSI may host or proxy its service through a network based SHPS or it may expose the service directly.
- **Provisioning** - modules (TMs, CSIs) can be provisioned over the wire to an advanced client. This functionality includes the full lifecycle support (provision, update, delete).

2.3. Modes of Operation

The Advanced Client will operate in multiple modes of operation based upon the parties that it is interacting with. The modes are differentiated by the connectivity level of the various actors within a transaction. The two primary modes of operation that we are concerned with here include:

- **Connected** - the Advanced Client is fully connected to the network and generally all parties to a transaction could communicate with each other if necessary. In this mode, the Advanced Client can choose to act as a simple facilitator of the actual operation or for various reasons (such as privacy, load balancing, etc.) the Advanced Client can take a more active role, providing delegated authentication and/or web services.
- **Disconnected** - the Advanced Client does not have connectivity to one or more parties in a transaction (such as not having connectivity to the IdP during an authentication transaction). This mode limits the Advanced Client ability to participate in facilitated operations and can restrict the availability of services exposed directly by the Advanced Client.

  Note that disconnected mode can also be used by the Advanced Client as a privacy preserving mode (where the IdP doesn't know exactly when a Advanced Client SSOs the user to a relying party, nor how many times).

2.4. Client Identification

The Advanced Client is an entity that may actively participate in transactions (such as being an WSC consuming a service or an issuer of assertions) and typically such entities are identified in Liberty ID-WSF protocols and SAML assertions by a ProviderID.

The choice of what to use as the ProviderID can lead to potential privacy issues as the ProviderID itself could be used as a correlation handle for the user sitting behind the Advanced Client (the same unique identifier showing up at many relying parties could be tracked and used by those parties to correlate independent behavior at each party).

Depending upon the privacy and security requirements for a particular implementation, the ProviderID MAY be one of the following:

- **unique identifier** - an identifier that identifies that specific Advanced Client uniquely amongst all other Advanced Clients (even on the same device).

  Using a unique identifier for the ProviderID of a Advanced Client opens the principal's privacy to potential correlation attacks and so it should not be used when privacy is of paramount importance.

  However, using a unique identifier does increase the traceability of operations (which can be a good thing from a security perspective) and it also allows an individual Advanced Client to be "canceled" without impacting other Advanced Clients issued by the same or other parties.

- **shared identifier** - an identifier that is shared amongst a group of Advanced Clients (typically manufactured by the same party).

  The definition of what makes up a "group" of Advanced Clients suitable for a shared identifier is up to the issuing party. A common solution is to use the same identifier across all instances of a Advanced Client issued by the same party with the same version of the software core for the Advanced Client (e.g., the same version of the Advanced Client).

  If the group is large enough in a particular context, the use of a **shared identifier** retains some of the security benefits of the **unique identifier** while at the same time retaining some of the privacy benefits of the **common identifier**.

  For example, if there's a problem with a particular version of the Advanced Client, it can be black-listed, requiring an upgrade to a fixed version prior to subsequent use), while also maintaining the privacy benefits of a common identifier as the correlation value is very small when there are many Advanced Clients with the same identifier.

- **common identifier** - a generic identifier assigned to all Advanced Clients that do not have a shared or unique identifier. Liberty has reserved the following URN for this case: *urn:liberty:idp:2007-09:ProviderID:Common*

  The common identifier eliminates the ProviderID as a potential correlation factor (although it doesn't get rid of other correlation handles such as the Advanced Client's IP address), but it also means that other means must be enacted to deal with security issues like revocation.

The Advanced Client's ProviderID should be used with strong caution in making security decisions unless it is somehow cryptographically protected in the transaction by the issuing party. One such way to protect the ProviderID is to include that provider ID in any security tokens issued to the Advanced Client to show that the token is associated with that ProviderID.

3. Identity Provider Operations

The TM can participate in a number of operations which are typically associated with an IdP. These include SSO and federation type operations.

3.1. SSO

3.1.1. Self-Asserted SSO

The TM may act as a full-fledged IdP, generating its own identities and assertions for those identities at SPs and WSPs. The TM may also expose an ID-WSF Discovery Service (DS) to allow the principal's services to be discovered and invoked.

Exposing such services is a double edged privacy sword for the TM. On one hand, the network based IdP has less vision into what the user is doing since they are not involved. However, on the other hand, the same TM is performing transactions at multiple SPs for a single (or very few) principals, making it more likely to be susceptible to correlation attacks by the SPs.

In order to expose these services, the TM need only follow the off-the-shelf protocols (Liberty AS, Liberty DS, SAML 2.0 Browser profile, etc.).

3.1.2. Facilitated SSO

Facilitated authentication takes place when the TM helps facilitate the authentication of the principal by using the *Enhanced Client or Proxy* profile in [SAMLProf2].

Facilitation does not make use of any of the enhanced capabilities of the TM. It is just listed here for completeness.

3.1.3. Delegated SSO

Delegated SSO takes place when the IdP delegates some the SSO process to the TM. The IdP is not directly involved at the time the principal SSOs to the SP/WSP. This essentially makes the TM an extension of the IdP, applying the appropriate security policies of the IdP when stepping through the SSO process at the SP/WSP.

3.1.3.1. Credentials for SSO

The TM needs appropriate credentials to present to a relying party for the purpose of authenticating the user to that party. There are two methods that can be used to place these credential(s) into the hands of the TM:

1. **minting** - the TM creates (or "mints") the necessary assertions when they are needed. With minting, only the credentials that are needed for a session are generated and they can be generated with more reasonable constraints (such as very small consumption periods).

   Minting does have a downside, though, in that many methods used to create trusted assertions end up leaking unique information that could be used as a correlation factor across multiple providers (such as the key used for a digital signature – if the same key is used to sign assertions for 2 different providers, the key itself can be a correlation factor).

   The correlation factor can be avoided in the case of minting by the TM using a different public key for each provider for which the TM generates keys **and** by ensuring that once a key is used with a provider, that key will not subsequently be used with any other provider.

2. **hoarding** - the TM stores (or "hoards") assertions for any potential party that they may communicate with and selects the one that is appropriate for the situation.

   Hoarding side-steps the correlation issue since each SP would receive an assertion generated and signed by the IdP itself (and presumably, the IdP has many principals, so SPs can't correlate one principal's actions). However, this is at substantial cost since the IdP has to generate assertions for every possible SP that the TM may interact with **and** those assertions have to have a long enough lifetime for the TM's purposes (i.e., they aren't created for immediate consumption).

The choice as to which method a TM will use is between the TM and the IdP and may be different for different combinations of TMs, IdPs and SPs.

3.1.4. Minting Assertions

The TM may sometimes need to create an assertion for consumption at an SP. We refer to this process as **minting** and such a created assertion as a **minted assertion** (MED). In order to mint a MED, the TM needs:

- **minting assertion** (MING) – An assertion issued by an IdP authorizing the TM to mint MEDs (typically with conditions limiting the applicability of the MEDs). See [LibertyIdP] for a detailed description of MINGs.
- **ProviderID** of the relying party(ies) for which the TM would like to generate assertions.
- **nameID** for the principal at the relying party(ies).

3.1.4.1. Obtaining Minting Credential

The TM uses the `<idp:GetAssertion>` interface exposed by the IdP to obtain minting assertions. To obtain minting assertions the TM should submit such a request with the following settings:

- Set the `assnType` attribute to:

  urn:liberty:idp:2007-09:purpose:minting
- Include a `<saml2p:AuthnRequest>` for each provider for which a MING is desired.
- If correlation across providers is a privacy concern, specify a unique public key for each `<saml2p:AuthnRequest>` in the `<saml2p:Subject>`'s `<saml2p:SubjectConfirmation>` element.

  To ensure non-correlation, once a public key is used with a given provider, the key should not be used with any other providers – ever.

  The TM is normally expected to generate private/public key pairs as necessary for minting assertions. It is possible that some environments may utilize some proprietary means of key management such that the IdP knows which keys to use for which provider and specifying them on this request would not be necessary in such environments.
- Specify the ProviderIDs for the providers in the `<saml2:Conditions>`'s `<saml2:AudienceRestriction>` element. If assertions for multiple providers are desired **AND** correlation is **NOT** a concern, multiple provider IDs can be specified within different `<saml2:Audience>` elements in the same `<saml2:AudienceRestriction>` element.

  If correlation is a concern, the MING for each provider should be requested within separate `<saml2p:AuthnRequest>` elements in the `<idp:GetAssertion>` element.
- If desired, specify the requested expiration time for the MING(s) using the `<saml2:Conditions>`'s `NotOnOrAfter` attribute.
- Other desired conditions may be specified including some of the conditions we added above. For example, the `<AuthnContextRestriction>` by the TM to specify the AuthnContexts it plans to use.

A sample `<idp:GetAssertion>` message:

```
<idp:GetAssertion purpose="urn:liberty:idp:2007-09:purpose:minting">

  <saml2p:AuthnRequest ID="ID_2343823023823" Version="2.0"
        IssueInstant="2006-06-23T15:38:46Z">
    <saml2:Subject>
      <saml2:SubjectConfirmation Method="...:holder-of-key">
        <saml2:SubjectConfirmationData>
          <ds:KeyInfo>Key info for Minting Assn (TM Public Key)</ds:KeyInfo>
        </saml2:SubjectConfirmationData>
      </saml2:SubjectConfirmation>
    </saml2:Subject>
    <saml2p:NameIDPolicy Format="...:persistent"/>
    <saml2:Conditions NotOnOrAfter="2006-07-23T15:38:46Z">
      <saml2:AudienceRestriction>
        <saml2:Audience>Provider 2</saml2:Audience>
      </saml2:AudienceRestriction>
    </saml2:Conditions>
  </saml2p:AuthnRequest>

</idp:GetAssertion>
```

Example 1. Example `<GetAssertion>` Message

3.1.4.2. Minting a Credential

The process of minting a credential follows the required processing in the SAML 2.0 specifications. The only added element here is that the `<saml2:Advice>` element MUST contain the MING. All other processing is pure SAML 2.0.

3.1.5. Hoarding Credentials

Hoarding credentials involves the TM obtaining relatively long lived SSO assertions issued by the IdP for consumption at a relying part. The TM decides when to obtain the credentials from the IdP and also determines when to provide them to the relying party.

Hoarding allows the TM to interact with relying parties without requiring real-time interaction with the IdP. This is sometimes desired due to privacy reasons and sometimes due to connectivity reasons.

Hoarded credentials are standard SAML 2.0 assertions. The only thing that makes them "hoarded" is that the TM requests them in advance of needing them (and hence they tend to have longer consumption lifetimes than the typical SAML Browser profile SSO Assertion).

3.1.5.1. Obtaining Credentials

The TM uses the `<idp:GetAssertion>` interface exposed by the IdP to obtain hoarding assertions. This is the same call used by the TM to obtain MINGs – only the `assnType` attribute is different.

To obtain hoarding assertions the TM should submit such a request with the following settings:

- Set the `assnType` attribute to:

  urn:liberty:idp:2007-09:purpose:sso

- Include an <saml2p:AuthnRequest> with the ProviderIDs for the desired relying party in the <saml2:Conditions>'s <saml2:AudienceRestriction> element.

  If assertions for multiple providers are desired, AND correlation across multiple providers is NOT a privacy concern multiple provider IDs can be specified within different <saml2:Audience> elements in the same <saml2:AudienceRestriction> element. If correlation is a concern, specify each relying party in separate <saml2p:AuthnRequest> elements.

  Even if multiple providers are specified in a single <saml2p:AuthnRequest> the IdP MAY return separate assertions for each provider.

- If desired, specify the requested expiration time for the assertion(s) using the <saml2:Conditions>'s NotOnOrAfter attribute.

- Other desired conditions, AuthnContexts, etc., may be specified.

A sample <idp:GetAssertion> message for obtaining MEDs:

```
<idp:GetAssertion purpose="urn:liberty:idp:2007-09:purpose:sso">

  <saml2p:AuthnRequest ID="ID_153282378" Version="2.0"
        IssueInstant="2006-06-23T15:38:46Z">
    <saml2p:NameIDPolicy Format="...:persistent"/>
    <saml2:Conditions NotOnOrAfter="2006-07-23T15:38:46Z">
      <saml2:AudienceRestriction>
        <saml2:Audience>Provider 2</saml2:Audience>
      </saml2:AudienceRestriction>
    </saml2:Conditions>
  </saml2p:AuthnRequest>

</idp:GetAssertion>
```

Example 2. Example <GetAssertion> Message

And an example response to that request:

```
<idp:GetAssertionResponse>
  <lu:Status code="OK"/>
  <idp:GetAssertionResponseItem ref="153282378">
    <idp:AssertionItem created="false">
      <saml2:Assertion>..Provider 2 Assertion.. </saml2:Assertion>
    </idp:AssertionItem>
  </idp:GetAssertionResponseItem>
</idp:GetAssertionResponse>
```

Example 3. Example <GetAssertionResponse> Message

3.1.6. Using Delegated SSO

Getting credentials (hoarded or minted) into the hands of the TM is only half of the problem. The credential must get into the hands of the relying party in order to be useful. This section talks about two common scenarios where the credentials can pass from the TM into the protocol flow to the relying party.

3.1.6.1. Browser Based SSO

A very common scenario where a web browser is able to use the TM as part of standardized Browser-based SSO protocols (in particular SAML's Browser-based SSO profile).

In order to use a TM, the browser would typically need an extension or plug-in which is able to:

- expose a SAML 2.0 Enhanced Client/Poxy (ECP) to relying parties (insert the necessary headers and process the incoming requests)
- communicate with the TM to obtain the necessary credential(s) to answer the incoming requests (treat the TM as the IdP).

This extension would listen for incoming requests and if present, process them, requesting the appropriate credentials from the TM and sending them back to the relying party in response.

3.1.6.2. SSO to WSPs

Applications running besides TMs (perhaps in the same computing environment) will want to use the TMs in order to authenticate to relying parties (WSPs). For example, a mail client may want to use the credentials available in a TM to authenticate with the mail server.

For ID-WSF based WSCs, the invocation of a WSP involves the step of discovering and then invoking the WSP. This process is typically bootstrapped by a browser based SSO session (where the DS-EPR is included in the SSO credential) or by the WSC invoking the Liberty AS to obtain the DS-EPR directly. It is also possible that the WSC is already in possession of the discovery information for the WSP and only needs a security token to invoke the WSP.

In either case, the TM can provide the credential necessary to communicate with the Liberty DS or the WSP.

3.2. Identity Federation

In this context, Federation is the creation and management of persistent identity handles which represent the identity of the user at the relying party. The fully qualified identity handle consists of a unique identifier, the issuer, and the relying party (in other words, it is only a handle for the user in this specific context and may be different in other contexts).

The creation of these handles raises potential race conditions when multiple parties (the TM and the IdP itself) may create a handle simultaneously for the same user at the same relying party.

3.2.1. Self-Asserted Identity Federations

Like with authentication assertions, the TM may act as a full-fledged IdP, generating its own identities that are used to federate the user's local identity (to the TM) to an identity at the relying party (the WSPs or SPs).

3.2.2. Online Federations

When the TM is able to connect to the IdP, we **strongly** recommend that the TM treat the federation event as a proxy event and use the IdP for the federation data (so that the IdP creates the federation handle at that time).

4. Client hosted services

Stronger and more capable client platforms has led to the desire for those platforms to be the primary host of one or more of a user's service instances. For example, a user may choose to host their primary contact book service on their PDA cellphone. This client hosted service is referred to as a Client Service Instance (CSI).

4.1. Client Service Components

The diagram below shows a typical set of interested parties in a client hosted service situation.

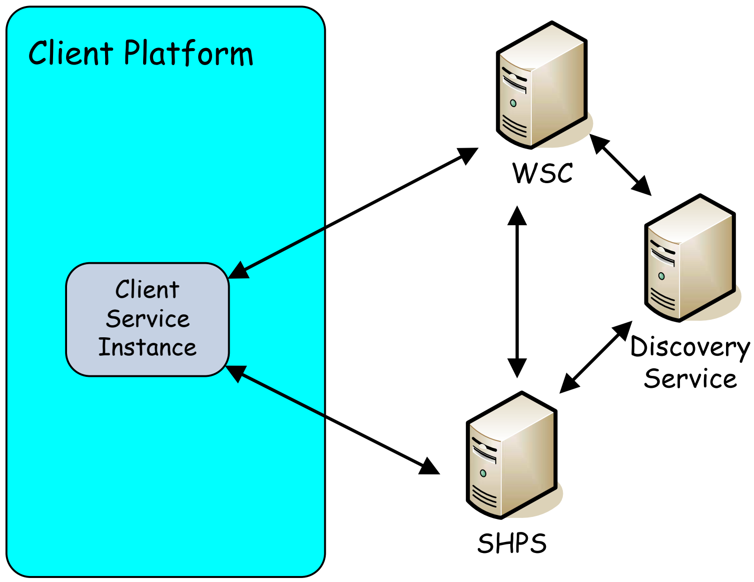

Figure 2. CSI Actors

The CSI is hosted on the client platform while the other parties, including the Web Services Consumer (WSC – the entity trying to invoke the service instance), Service Hosting/Proxying Service (SHPS) and the Discovery Service (DS), are typically applications hosted on network servers.

The Service Hosting/Proxying Service provides a means for the CSI to use a remote, network visible entity to expose/proxy its service. We'll talk more about that shortly.

The Discovery Service comes into play when discussing CSIs because service instances must be registered in the DS in order for them to be found by the WSC. Several of the operations surrounding the SHPS enablement will involve DS interactions.

This complexity is necessitated by a number of issues that must be considered when hosting a service on the client, including:

- Clients frequently have limited communications bandwidth (when compared to online services).
- Clients have more tenuous connectivity (being unavailable when the user goes through a tunnel or turns the device off for the night).
- Clients are frequently behind network firewalls, preventing incoming service invocation without modifications to the firewall rules.
- Multiple providers talking to the same service endpoint on the client for a particular user's service can use that service endpoint as a correlation handle for the user and potentially collude without user knowledge or control.

The user, and their client service instance, have the choice of the following hosting solutions:

- **Stand-alone** - the service is hosted exclusively on the client device and all service invokers must communicate directly with the device. This is the standard ID-WSF web services provider model where the service instance maintains a service metadata description at the Liberty ID-WSF Discovery Service (DS). Web service consumers (WSCs) discover and invoke the service instance using the Discovery Service.

  Such implementations must deal with or accept the connectivity and privacy issues outlined above. The PAOS protocol (see [LibertyPAOS] may be used with stand-alone service instances to resolve some of the connectivity issues.

  In most cases the stand-alone solution is used where collusion protection is not a concern and the service instance is on an always-connected device exposed directly on an external network.

- **Proxied** - the service instance is hosted exclusively on the client device, but uses the Liberty ID-WSF Service Hosting/Proxying service (SHPS) to proxy incoming calls. This solves two of the problems listed above: a) the privacy breaking cross-provider collusion concern is mitigated by the large number of client using the same SHPS service and b) the client doesn't have to have an externally exposed interface as it can poll the SHPS service for incoming request.

  In this case, the SHPS would be registered as the endpoint for the service instance for the user in the DS. WSCs would invoke the service instance at the SHPS and the SHPS would forward the request to the client, get the response back and forward the response to the WSC. The WSC would not be aware that the proxying is taking place.

  If the service proxy hosted at SHPS is invoked when the client instance is not available, the call fails as in this mode the SHPS is not configured to act in the name of the client.

- **Hosted** - a mirror of the service instance is hosted on the SHPS. Requests for service are handled directly by the hosted instance without additional interaction with the client instance. The client service instance keeps the hosted mirror service instance up-to-date as necessary.

  In this case, the SHPS would be registered as the endpoint for the service instance for the user in the DS. WSCs would invoke the service instance at the SHPS and SHPS would respond directly without involving the client.

- **Proxied+Hosted** - both hosting and proxying are implemented. Proxying is used when the client is available and when the client is not available SHPS is able to respond to the request using the data in it's mirrored service instance.

4.2. Services Data

The following objects are defined by the client services system:

- **Service Handle** (SH) - a reference handle to the hosted/proxied service instance at the SHPS. This is assigned by SHPS during service registration and used whenever the client needs to interact directly with SHPS with regard to this service instance.

- **Service Descriptor** - a profiled ID-WSF EPR which contains a description of a service that the SHPS hosts and/or proxies.

4.3. SHPS Examples

The following examples will show a few of the many workflows enabled by the SHPS. We will first walk through the registration of a service instance at the SHPS and later through the use of that service instance.

4.3.1. Hosted Service Instance Registration

This use case walks through a potential sequence of steps to register and enable a hosted service instance of the Liberty ID-SIS Personal Identity Profile at the SHPS.

The sequence diagram below shows the sequence of steps between the various components:

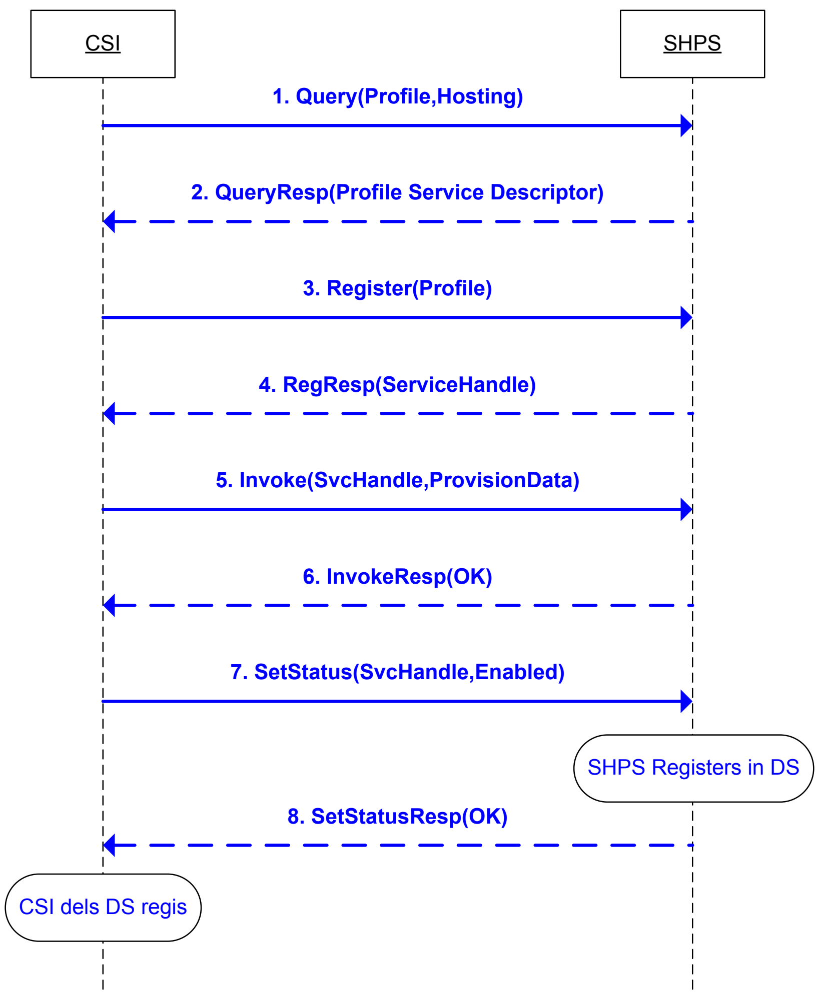

Figure 3. Hosted Service Instance Registration Workflow

Things to note about the diagram:

- The CSI exists on some platform and initiates the registration process due to some undefined event (perhaps the user selecting an option on an administrative interface).
- How the CSI figures out which SHPS to talk to is out of scope for this example. In most cases the SHPS will be discovered using the Liberty ID-WSF protocols (either because it's a publicly visible SHPS or because the user has an identity relationship with the SHPS).

4.3.1.1. Step 1: Query for supported service info

The CSI uses the `<shps:Query>` interface to ask the SHPS for the details on the support of a hosted Profile service.

```
<shps:Query>
  <disco:RequestedService>
    <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
    <disco:SecurityMechID>urn:liberty:sm:2006-08:TLS:SAMLV2</disco:SecurityMechID>
    <disco:Framework version="2.0" />
    <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:hosted</shps:ServiceMode>
  </disco:RequestedService>
</shps:Query>
```

Example 4. `<shps:Query>` Request Message

4.3.1.2. Step 2: SHPS responds with service info

The SHPS responds with a `<shps:QueryResponse>` message indicating that it can support a hosted service instance of the Liberty ID-SIS Personal Profile service.

```
<shps:QueryResponse>
  <lu:Status code="OK" />
  <wsa:EndpointReference>
    <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
    <wsa:Metadata>
      <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:hosted</shps:ServiceMode>
      <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
      <disco:Framework version="2.0" />
      <disco:SecurityContext>
        <disco:SecurityMechID>urn:liberty:security:2005-02:TLS:SAML</disco:SecurityMechID>
      </disco:SecurityContext>
    </wsa:Metadata>
  </wsa:EndpointReference>
</shps:QueryResponse>
```

Example 5. `<shps:QueryResponse>` Request Message

4.3.1.3. Step 3: Register Hosted Service Instance

The CSI submits a registration request for the profile service.

```
<shps:Register>
  <wsa:EndpointReference lu:itemID="1">
    <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
    <wsa:Metadata>
      <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:hosted</shps:ServiceMode>
      <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
      <disco:Framework version="2.0" />
      <disco:SecurityContext>
        <disco:SecurityMechID>urn:liberty:security:2006-08:TLS:SAMLV2</disco:SecurityMechID>
      </disco:SecurityContext>
   </wsa:Metadata>
  </wsa:EndpointReference>
</shps:Register>
```

Example 6. `<shps:Register>` Request Message

4.3.1.4. Step 4: Registration response from SHPS

The SHPS responds with a successful status code and the assigned service handle for this new service instance.

```
<shps:RegisterResponse>
  <lu:Status code="OK" />
  <shps:RegisterResponseItem ref="1">
    <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
  </shps:RegisterResponseItem>
</shps:RegisterResponse>
```

Example 7. `<shps:RegisterResponse>` Request Message

4.3.1.5. Step 5: Initialize data in hosted service

The CSI uses the `<shps:Invoke>` interface to invoke the service instance's interfaces to initialize the service with the necessary data to enable hosting of the service.

This step may be invoked multiple times if several invocations are needed to setup the hosted data. It may also be invoked at other times after the service has been enabled to update and/or query the data for the hosted service instance.

```
<shps:Invoke>
  <shps:InvokeItem itemID="1">
    <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
    <pp:Modify>
      <pp:ModifyItem>
          ... modification data goes here ...
      </pp:ModifyItem>
    </pp:Modify>
  </shps:InvokeItem>
</shps:Invoke>
```

Example 8. `<shps:Invoke>` Request Message

4.3.1.6. Step 6: Initialization response

The SHPS responds to the invocation request. This is a successful invocation response which contains a personal profile modification response. Note that the status of the personal profile modification request may be `Failed` even though the status of the invocation response is `OK`. The former is the status of the actual service request while the latter is the status of whether or not the service was invoked.

```
<shps:InvokeResponse>
  <lu:Status code="OK" />
  <shps:InvokeResponseItem ref="1">
    <pp:ModifyResponse>
          ... modification response data goes here ...
    </pp:ModifyResponse>
  </shps:InvokeResponseItem>
</shps:InvokeResponse>
```

Example 9. `<shps:InvokeResponse>` Request Message

4.3.1.7. Step 7: Enable the service instance

The CSI uses the `<shps:SetStatus>` interface to enable the service instance on the SHPS. Enabling a service instance causes the SHPS to register the service instance in the user's Discovery Service (so that the SHPS service instance is discoverable by WSCs).

```
<shps:SetStatus>
  <shps:SetStatusItem itemID="1" >
    <shps:ServiceStatus>urn:liberty:shps:2007-09:status:enabled</shps:ServiceStatus>
    <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
  </shps:SetStatusItem>
</shps:SetStatus>
```

Example 10. `<shps:SetStatus>` Request Message

4.3.1.8. Step 8: Service Enablement Response

The SHPS responds to the enablement request. This is a successful response.

Typically, after the successful enablement of the hosted service instance, the CSI will de-register its service instance from the user's Discovery Service (if it had been registered). This is to prevent potential problems if WSCs were to see both the SHPS and CSI instances of the service.

```
<shps:SetStatusResponse>
  <lu:Status code="OK" />
</shps:SetStatusResponse>
```

Example 11. `<shps:SetStatusResponse>` Request Message

4.3.2. Proxied Service Instance Registration

This use case walks through a potential sequence of steps to register and enable a proxied service instance of the Liberty ID-SIS Personal Identity Profile at the SHPS.

This example is very similar to the Hosted Service Registration example (see Section 4.3.1). The difference being the type of data registered and the lack of need of initializing the data in the hosted service instance.

The sequence diagram below shows the sequence of steps between the various components:

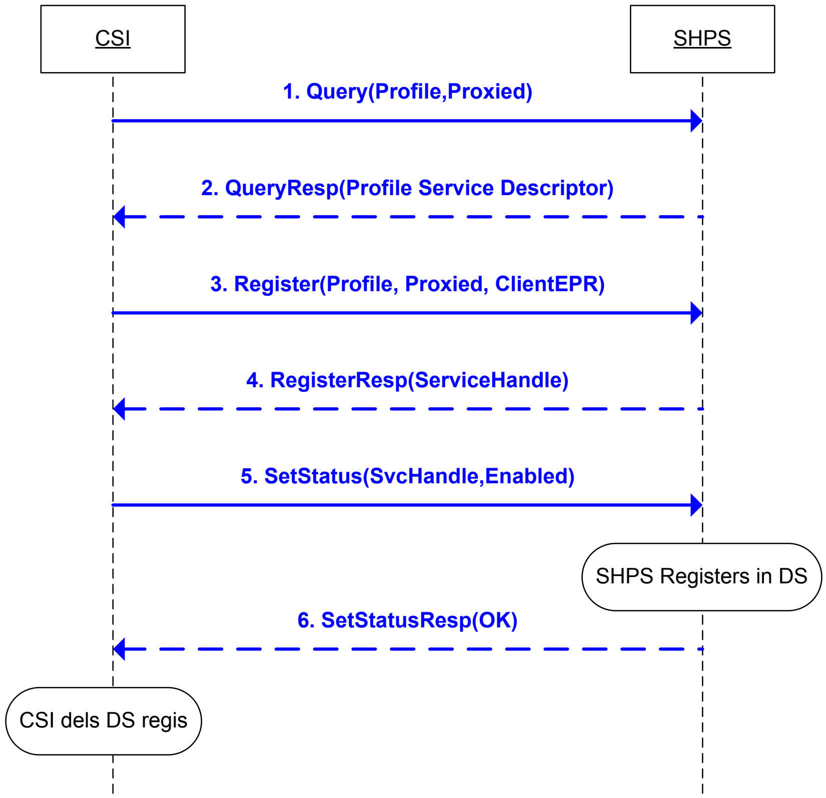

Figure 4. Proxied Service Instance Registration Workflow

Things to note about the diagram:

- The CSI exists on some platform and initiates the registration process due to some undefined event (perhaps the user selecting an option on an administrative interface).
- How the CSI figures out which SHPS to talk to is out of scope for this example. In most cases the SHPS will be discovered using the Liberty ID-WSF protocols (either because it's a publicly visible SHPS or because the user has an identity relationship with the SHPS).

4.3.2.1. Step 1: Query for supported service info

The CSI uses the `<shps:Query>` interface to ask the SHPS for the details on the support of a proxied Profile service.

```
<shps:Query>
  <disco:RequestedService>
    <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
    <disco:SecurityMechID>urn:liberty:sm:2006-08:TLS:SAMLV2</disco:SecurityMechID>
    <disco:Framework version="2.0" />
    <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:proxied</shps:ServiceMode>
  </disco:RequestedService>
</shps:Query>
```

Example 12. `<shps:Query>` Request Message

4.3.2.2. Step 2: SHPS responds with service info

The SHPS responds with a <shps:QueryResponse> message indicating that it can support a proxied service instance of the Liberty ID-SIS Personal Profile service.

```
<shps:QueryResponse>
  <lu:Status code="OK" />
  <wsa:EndpointReference>
    <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
    <wsa:Metadata>
      <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:proxied</shps:ServiceMode>
      <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
      <disco:Framework version="2.0" />
      <disco:SecurityContext>
        <disco:SecurityMechID>urn:liberty:security:2005-02:TLS:SAML</disco:SecurityMechID>
      </disco:SecurityContext>
    </wsa:Metadata>
  </wsa:EndpointReference>
</shps:QueryResponse>
```

Example 13. `<shps:QueryResponse>` Request Message

4.3.2.3. Step 3: Register Proxied Service Instance

The CSI submits a registration request for the profile service with an anonymous <shps:CallbackEPR> indicating that the client will poll for requests.

```
<shps:Register>
  <wsa:EndpointReference lu:itemID="1">
   <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
   <wsa:Metadata>
     <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:proxied</shps:ServiceMode>
     <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
     <disco:Framework version="2.0" />
     <disco:SecurityContext>
       <disco:SecurityMechID>urn:liberty:security:2006-08:TLS:SAMLV2</disco:SecurityMechID>
     </disco:SecurityContext>
     <shps:CallbackEPR>
       <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
     </shps:CallbackEPR>
   </wsa:Metadata>
  </wsa:EndpointReference>
</shps:Register>
```

Example 14. `<shps:Register>` Request Message

4.3.2.4. Step 4: Registration response from SHPS

The SHPS responds with a successful status code and the assigned service handle for this new service instance.

```
<shps:RegisterResponse>
  <lu:Status code="OK" />
  <shps:RegisterResponseItem ref="1">
    <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
  </shps:RegisterResponseItem>
</shps:RegisterResponse>
```

Example 15. `<shps:RegisterResponse>` Request Message

4.3.2.5. Step 5: Enable the service instance

The CSI uses the `<shps:SetStatus>` interface to enable the service instance on the SHPS. Enabling a service instance causes the SHPS to register the service instance in the user's Discovery Service (so that the SHPS service instance is discoverable by WSCs).

```
<shps:SetStatus>
  <shps:SetStatusItem itemID="1" >
    <shps:ServiceStatus>urn:liberty:shps:2007-09:status:enabled</shps:ServiceStatus>
    <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
  </shps:SetStatusItem>
</shps:SetStatus>
```

Example 16. `<shps:SetStatus>` Request Message

4.3.2.6. Step 6: Service Enablement Response

The SHPS responds to the enablement request. This is a successful response.

Typically, after the successful enablement of the proxied service instance, the CSI will de-register its service instance from the user's Discovery Service (if it had been registered). This is to prevent potential problems if WSCs were to see both the SHPS and CSI instances of the service.

```
<shps:SetStatusResponse>
  <lu:Status code="OK" />
</shps:SetStatusResponse>
```

Example 17. `<shps:SetStatusResponse>` Request Message

4.3.3. Proxied Service Invocation

This use case walks through the invocation of a proxied service by a WSC.

The sequence diagram below shows the sequence of steps between the various components:

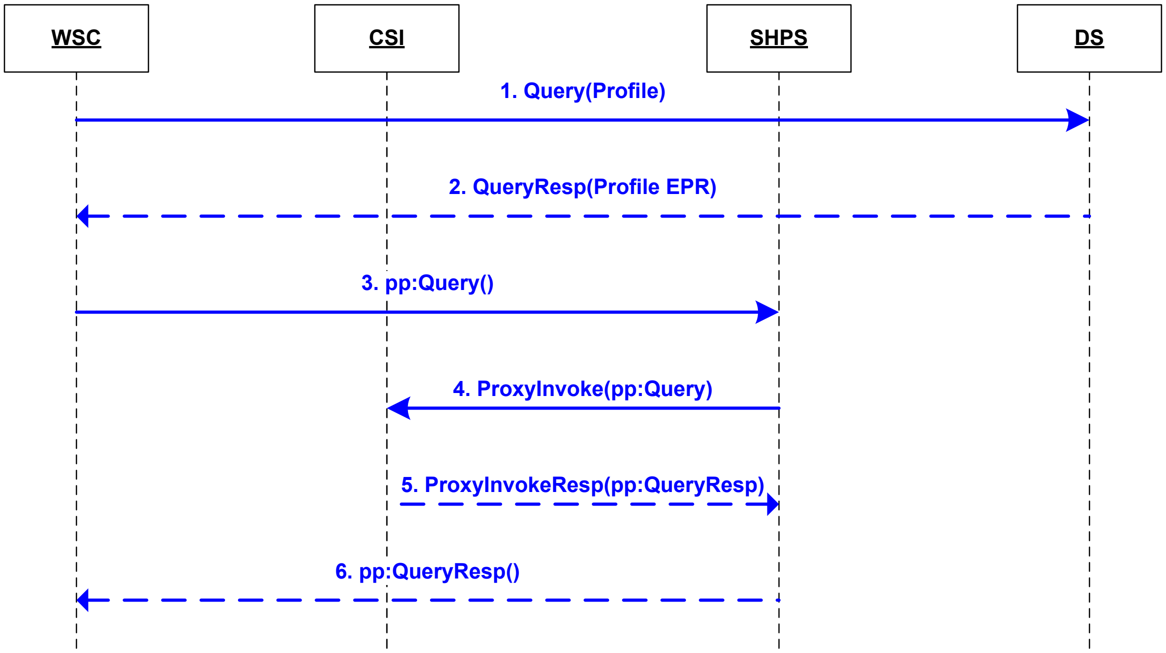

Figure 5. Proxied Service Invocation Workflow

Things to note about the diagram:

- The service instance has already been registered and enabled at the SHPS at some point in the past.
- The first steps (Discovery) are exactly the same as they would be for a normal service invocation (as well as for a SHPS hosted service instance).
- The CSI has registered an CallbackEPR with the SHPS that allows direct invocation rather than requiring polling from the CSI.

4.3.3.1. Step 1: Discover the profile service provider

The WSC requests the service endpoint for the Profile Service from the discovery service. This is a standard invocation of the Discovery Service by the WSC – there is nothing special about the request for this use case.

```
<disco:Query>
  <disco:RequestedService>
    <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>

    <disco:SecurityMechID>urn:liberty:security:2006-08:ClientTLS:SAMLV2</disco:SecurityMechID>
    <disco:SecurityMechID>urn:liberty:security:2005-02:ClientTLS:SAML</disco:SecurityMechID>
    <disco:SecurityMechID>urn:liberty:security:2006-08:TLS:SAMLV2</disco:SecurityMechID>
    <disco:Framework version="2.0" />
  </disco:RequestedService>
</disco:Query>
```

Example 18. `<disco:Query>` Request Message

4.3.3.2. Step 2: DS responds with service EPR

The DS responds with a `<disco:QueryResponse>` message which includes the Profile Service EPR.

```
<disco:QueryResponse>
  <lu:Status code="OK" />
  <wsa:EndpointReference disco:notOnOrAfter="2007-12-08T13:23:40Z" >
    <wsa:Address>http://shps.services.com/pp</wsa:Address>
    <wsa:Metadata>
      <disco:Abstract>Bob's personal profile</disco:Abstract>
      <disco:ProviderID>http://services.com/</disco:ProviderID>
      <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
      <disco:Framework version="2.0" />
      <disco:SecurityContext>
        <disco:SecurityMechID>urn:liberty:security:2005-02:TLS:SAMLV2</disco:SecurityMechID>
        <sec:Token usage="urn:liberty:security:tokenusage:2006-08:SecurityToken">
          <saml2:Assertion  ...>
            .... SAML Token data goes here ....
          </saml2:Assertion>
        </sec:Token>
      </disco:SecurityContext>
    </wsa:Metadata>
  </wsa:EndpointReference>
</disco:QueryResponse>
```

Example 19. `<disco:QueryResponse>` Request Message

4.3.3.3. Step 3: WSC Invokes Profile service at SHPS

The WSC submits a Profile service request to the profile service provider (the WSC does not know that the profile service provider is a SHPS that is proxying the request).

```
<pp:Query>
   ... query data goes here ...
</pp:Query>
```

Example 20. `<pp:Query>` Request Message

4.3.3.4. Step 4: Proxied invocation of CSI from SHPS

The SHPS uses the `<shps:ProxyInvoke>` interface on the CSI to pass along the `<pp:Query>` request.

```
<shps:ProxyInvoke>
  <shps:ProxyInvokeItem itemID="1">
    <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
    <shps:InvocationContext>
      <shps:InvokingProvider>http://services.corp.com</shps:InvokingProvider>
      <disco:SecurityMechID>urn:liberty:security:2006-08:TLS:SAMLV2</disco:SecurityMechID>
    </shps:InvocationContext>
    <pp:Query>
       ... query data goes here ...
    </pp:Query>
  </shps:ProxyInvokeItem>
</shps:ProxyInvoke>
```

Example 21. `<shps:ProxyInvoke>` Request Message

4.3.3.5. Step 5: Proxied Response from CSI to SHPS

The CSI places the service response within the <shps:ProxyInvokeResponse> message.

```
<shps:ProxyInvokeResponse>
  <lu:Status code="OK" />
  <shps:ProxyInvokeResponseItem ref="1">
    <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
    <shps:ResponseHeaders>
      <sb:UsageDirectives> .....  </sb:UsageDirectives>
    </shps:ResponseHeaders>
    <pp:QueryResponse>
          ... query response data goes here ...
    </pp:QueryResponse>
  </shps:ProxyInvokeResponseItem>
</shps:ProxyInvokeResponse>
```

Example 22. <shps:ProxyInvokeResponse> Request Message

4.3.3.6. Step 6: SHPS passes response to WSC

The SHPS copies the service response from the proxied response to its response message to the WSC.

```
<pp:QueryResponse>
  ... query response data goes here ...
</pp:QueryResponse>
```

Example 23. <pp:QueryResponse> Request Message

5. Provisioning

Provisioning, in this context, refers to the distribution, installation and maintenance (update/delete) of some functional module (perhaps a TM) onto a device or platform. The specific capabilities and features of a particular functional module are out of scope here. This process is only concerned with getting the functional module up and running within the target environment. So, a functional module may be a TM that is an extension of the IdP performing delegated SSO operations, or the module may be a TM that is nothing more than an active participant in traditional SSO operations (a-la the SAML 2.0 Enabled Client/Proxy).

5.1. Provisioning Components

The following diagram illustrates the components involved in the provisioning process:

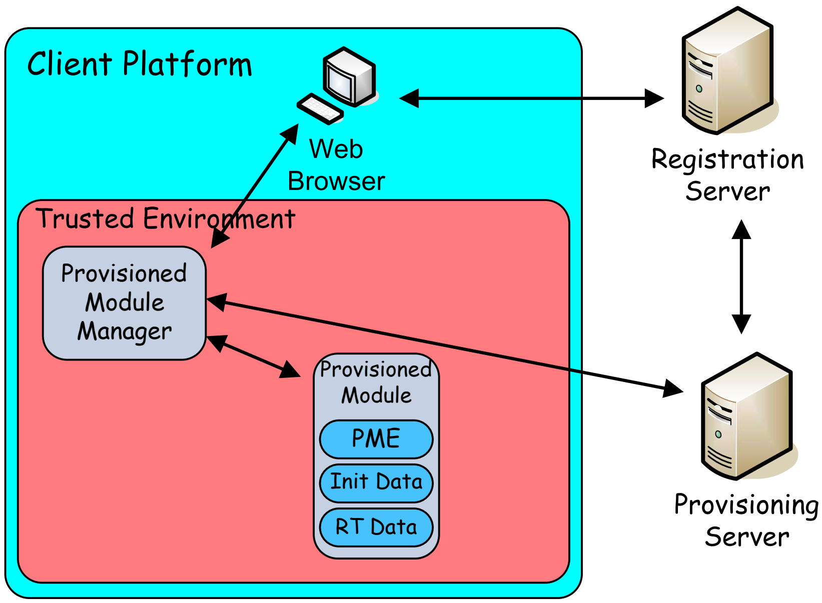

Figure 6. Provisioning Components

Things to note about this diagram:

- It is not drawn to any form of scale!
- The client platform represents any type of client, such as a personal computer, a device, a smart card, etc..
- The trusted environment represents some form of tamper resistant container (thus providing a level of trust for the provisioned components). The trusted environment is **not** a requirement of these protocols – the components shown within the trusted environment could very well exist directly within the relatively untrusted client platform (e.g. the Provisioned Module Manager could run as a service within the Client Platform operating system).
- The Provisioned Module Manager (PMM) is a service running on the client platform which provides a beach head for provisioning operations. The PMM exposes the interfaces documented within the Liberty ID-WSF Provisioned Module Manager Service Specification [LibertyPMM].

  This document does not address the chicken-vs-egg issue of how the PMM comes into being on the client. It may be built into the platform or it may be manually installed by some party (such as the user). That discussion is out-of-scope.

- The Provisioned Module (PM) is a component which performs some set of functionality. For example, a PM could be a TM (a module which provides IdP extension functionality). PMs may also expose functionality that is not defined by Liberty specifications.

  Each PM is identified using a globally unique identifier called the Provisioned Module IDentifier (PMID). The PMID is used to reference to specific instances of a PM when performing tasks like status updates or module updates.

  The PM is shown as being composed of 3 distinct parts:

  - **Provisioned Module Engine (PME)** - the executable code which provides the functionality for the PM.

    This is defined as a separate component here to enable a provisioning process which allows the PME to preexist in the client platform and so just delivers the data necessary to instantiate the PM using that preexisting engine. Of course, the PME may not preexist and in such cases the PMM will have to retrieve it.

    During provisioning, the PME is passed by reference (name) so that the PMM can determine whether or not the PME already exists (either because it was pre-installed or because the same PME has been previously provisioned). Should the PMM need to obtain the PME, the passed in reference is used to identify the PME being downloaded.

  - **Initialization Data (PMInitData)** - the data needed by the PME in order to initialize a new instance of a PM. This may be the actual data needed by the PME or it may be a reference that the PME knows how to dereference and obtain the initialization data at runtime. This data may or may not be needed during the provisioning process. Some PMs are fully individualized and have their PMInitData built in.

    The format and structure of the PMInitData is out of scope for this document and is specific to the PME. It is up to the Provisioning Service to resolve what data is needed for what PME. The PMM treats PMInitData as an opaque data set that it passes to the PME upon initialization.

  - **Runtime Data (PMRTData)** - the runtime data created/managed by the PM instance as it performs its tasks. This would include things like MINGs for a TM that is minting assertions, private keys, etc. This is defined separate from the InitData to allow for PM portability (where a previously activated PM is moved to another client platform).

- The Web Browser in this diagram represents an enhanced browser (either directly or via a plug-in) with support for the provisioning process. In other provisioning use cases this may be an application or even be the PMM itself can instigate a new provision operation (typically via some direct interaction with the user).

- The Registration Server (RegS) is not a Liberty defined entity, but rather a deployment component for a particular set of use cases. In this use case, the RegS interacts with the user through a web browser and then controls the provisioning process using the interfaces on the Provisioning Server.

- The Provisioning Server (ProvS) is typically a network hosted service that is the primary entity with which the PMM interacts. This server is an instance of a Liberty ID-WSF Provisioning Service (see [LibertyPROV]).

  The primary function of the ProvS is to provide a trusted endpoint for the management and distribution of PMs.

5.2. Provisioning Data

The following objects are defined by the provisioning system:

- **Provisioning Handle** (PH) - a small data structure handed to the PMM to initiate the provisioning process. This will contain the location of, and instructions on how to invoke, the ProvS as well as a unique token (an artifact) to represent the PM that is to be provisioned.

- **Provisioned Module Descriptor** (PMD) - a data structure describing the components of the PM, including a reference to the necessary PME as well as any PMInitData and PMRTData for the PM.

5.3. Provisioning Examples

The provisioning system defined by Liberty allows for many different workflows to support many different implementations. We will walk through two different scenarios to show how the different components can work together to support the provisioning process.

5.3.1. Initial Provisioning

This use case walks through a potential sequence of steps to provision a new PM onto a client platform. In this case, the PMM is built into the platform and the user initiates the provisioning process within an enhanced web browser (RegApp) that is interacting with a Registration Service (RegS).

The RegS is the controlling entity in this use case, building and issuing the PMs (and using the ProvS as the distribution entity).

The sequence diagram below shows the sequence of steps between the various components:

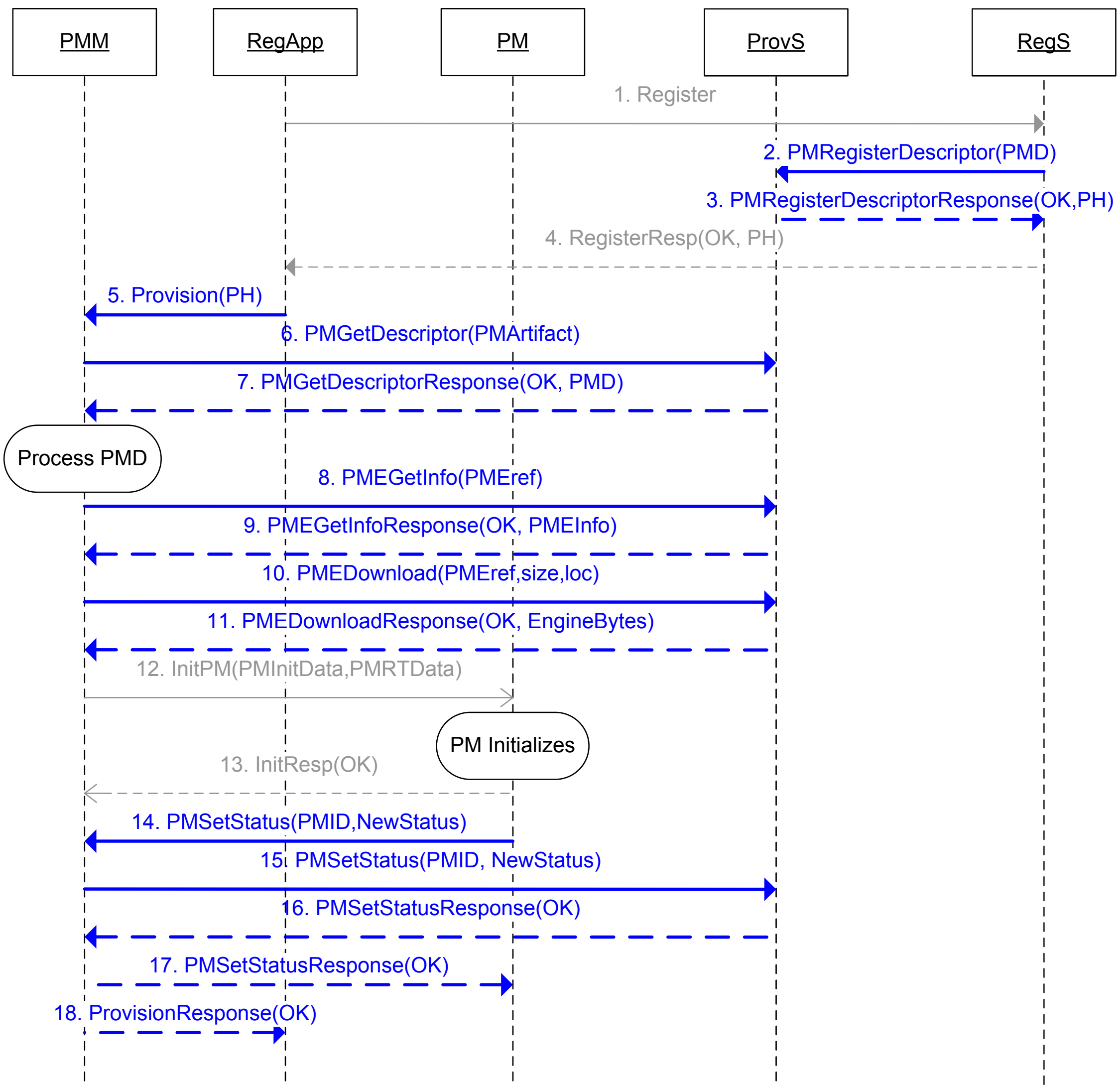

Figure 7. Provisioning Workflow

Things to note about the diagram:

- The steps shown in gray are not messages defined by any Liberty ID-WSF specification at this point in time. These messages are included to show a complete use case in some possible deployments.

- The PMM, RegApp, and PM all operate on a client platform. The RegApp is a registration application (or in some cases it could be a browser plug-in). The ProvS and RegS are typically network hosted services. Neither the RegApp, nor the RegS are Liberty defined entities.

5.3.1.1. Step 1: User registration.

The user initiates a registration operation at the RegS. This may take several messages and may involve the user creating a new relationship with the RegS or may involve a user with a previous relationship authenticating with the RegS.

These messages are not defined by Liberty ID-WSF specifications.

5.3.1.2. Step 2: PMD Registration at ProvS.

The RegS creates a PMD and uses the `<prov:PMRegisterDescriptor>` interface to register the PMD at the ProvS. This request would look something like:

```
<prov:PMRegisterDescriptor>
  <prov:PMRegisterDescriptorItem itemID="1">
    <prov:PMDescriptor xs:id="2323923900239">
      <prov:PMID issuer="http://provs-r-us.com">uuid:239032-230328-92379-2397923</prov:PMID>
      <prov:PMEngineRef>https://pmsRus.org/VeryTrustedModule/3.7</prov:PMEngineRef>
      <prov:PMInitData>
        <MyData>
          .... initialization data here ...
        </MyData>
      </prov:PMInitData>
      <ds:Signature>
        ... signature data goes here ...
      </ds:Signature>
    </prov:PMDescriptor>
  </prov:PMRegisterDescriptorItem>
</prov:PMRegisterDescriptor>
```

Example 24. `<prov:PMRegisterDescriptor>` Request Message

5.3.1.3. Step 3: ProvS responds to RegS with PH.

The ProvS processes the PMD and builds a PH which represents the PMD and returns it to the RegS. This response would look something like:

```
<prov:PMRegisterDescriptorResponse>
  <lu:Status code="OK" />
  <prov:PMRegisterDescriptorResponseItem ref="1">
    <prov:ProvisioningHandle xs:id="2302384823023">
      <prov:PMDArtifact>23asdfhoi323hposdf923h9sdfhweorh2398asdfjweoiha</prov:PMDArtifact>
      <prov:ProvisioningServiceEPR>
        <wsa:Address>http://provision.idpsRus.com</wsa:Address>
        <wsa:Metadata>
          <ds:Abstract>Provisioning Service</ds:Abstract>
          <ds:ProviderID>http://provisioning-provider.idpsRus.com/</ds:ProviderID>
          <ds:ServiceType>urn:liberty:prov:2007-09</ds:ServiceType>
          <ds:Framework version="2.0" />
          <ds:SecurityContext>
            <ds:SecurityMechID>
              urn:liberty:security:2005-02:TLS:SAMLV2
            </ds:SecurityMechID>
            <sec:Token ref="urn:liberty:disco:tokenref:ObtainFromIDP" />
          </ds:SecurityContext>
        </wsa:Metadata>
      </prov:ProvisioningServiceEPR>
      <ds:Signature>
        ... signature info here ..
      </ds:Signature>
    </prov:ProvisioningHandle>
  </prov:PMRegisterDescriptorResponseItem>
</prov:PMRegisterDescriptorResponse>
```

Example 25. `<prov:PMRegisterDescriptorResponse>` Message

5.3.1.4. Step 4: RegS sends PH to RegApp.

The RegS sends the PH to the RegApp.

This message is not defined by Liberty ID-WSF specifications.

5.3.1.5. Step 5: RegApp initiates Provisioning at PMM.

The RegApp uses the `<pmm:PMProvision>` interface to pass in the PH and initiate the provisioning process at the PMM. This request would look something like:

```
<pmm:Provision wait="true" >
  <prov:ProvisioningHandle xs:id="2302384823023">
    <prov:PMDArtifact>23asdfhoi323hposdf923h9sdfhweorh2398asdfjweoiha</prov:PMDArtifact>
    <prov:ProvisioningServiceEPR>
      <wsa:Address>http://provision.idpsRus.com</wsa:Address>
      <wsa:Metadata>
        <ds:Abstract>Provisioning Service</ds:Abstract>
        <ds:ProviderID>http://provisioning-provider.idpsRus.com/</ds:ProviderID>
        <ds:ServiceType>urn:liberty:prov:2007-09</ds:ServiceType>
        <ds:Framework version="2.0" />
        <ds:SecurityContext>
          <ds:SecurityMechID>
            urn:liberty:security:2005-02:TLS:SAMLV2
          </ds:SecurityMechID>
          <sec:Token ref="urn:liberty:disco:tokenref:ObtainFromIDP" />
        </ds:SecurityContext>
      </wsa:Metadata>
    </prov:ProvisioningServiceEPR>
    <ds:Signature>
      ... signature info here ..
    </ds:Signature>
  </prov:ProvisioningHandle>
</pmm:Provision>
```

Example 26. `<pmm:PMProvision>` Request Message

5.3.1.6. Step 6: PMM Asks ProvS for the PMD associated with PH.

The PMM parses the PH and uses the ProvS ID-WSF EPR within the PH to invoke the `<prov:PMArtifactResolve>` interface to exchange the PMArtifact that was in the PH for a PMD. This request would look something like:

```
<prov:PMGetDescriptor>
  <prov:PMDArtifact>23asdfhoi323hposdf923h9sdfhweorh2398asdfjweoiha</prov:PMDArtifact>
  <prov:CallbackEPR>
    <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
  </prov:CallbackEPR>
</prov:PMGetDescriptor>
```

Example 27. `<prov:PMGetDescriptor>` Request Message

5.3.1.7. Step 7: ProvS responds to the PMM with the PMD.

The ProvS processes the request, verifies the artifact and that has not been previously used, locates the matching PMD and returns the PMD to the PMM. This response would look something like:

```
<prov:PMGetDescriptorResponse>
  <lu:Status code="OK" />
  <prov:PMDescriptor xs:id="2323923900239">
    <prov:PMID issuer="http://provs-r-us.com">uuid:239032-230328-92379-2397923</prov:PMID>
    <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/3.7</prov:PMEngineRef>
    <prov:PMInitData>
      <MyData>
        .... initialization data here ...
      </MyData>
    </prov:PMInitData>
    <ds:Signature>
      ... signature data goes here ...
    </ds:Signature>
  </prov:PMDescriptor>
</prov:PMGetDescriptorResponse>
```

Example 28. `<prov:PMGetDescriptorResponse>` Message

5.3.1.8. Step 8: PMM retrieves PME information from the ProvS.

The PMM parses the PMD and noting that it does not have an instance of that PME invokes the `<prov:PMEGetInfo>` interface to obtain the PME information (size, hash, etc).

This request would **not** be necessary if the PMM already had an instance of the PME available locally (perhaps because another PM was instantiated which used the same PME, or because that particular PME was built into the platform).

This request would look something like:

```
<prov:PMEGetInfo>
  <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
</prov:PMEGetInfo>
```

Example 29. `<prov:PMEGetInfo>` Request Message

5.3.1.9. Step 9: ProvS responds to the PMM with the PME Info.

The ProvS returns the PME Information to the PMM. This response would look something like:

```
<prov:PMEGetInfoResponse>
  <lu:Status code="OK" />
  <prov:PMEInfo>
    <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
    <prov:PMECreatorID>http://reg.providers.com"</prov:PMECreatorID>
    <prov:PMEWhenCreated>2007-01-18T17:32:14Z</prov:PMEWhenCreated>
    <prov:PMEEnabled>true</prov:PMEEnabled>
    <prov:PMEWhenEnabled>2007-01-18T18:15:22Z</prov:PMEWhenEnabled>
    <prov:PMESize>185676</prov:PMESize>
    <prov:PMEHash method="...SHA-1">...SHA1 hash data ...</prov:PMEHash>
  </prov:PMEInfo>
</prov:PMEGetInfoResponse>
```

Example 30. `<prov:PMEGetInfoResponse>` Message

5.3.1.10. Step 10: PMM retrieves the PME from the ProvS.

The PMM initiates the download of the PME from the ProvS using the `<prov:PMEDownload>` interface.

This PMM specifies how much of the PME it wants to get in each `<prov:PMEDownload>` invocation and will usually invoke this several times to download the entire executable.

This request would look something like:

```
<prov:PMEDownload count="102400">
  <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
</prov:PMEDownload>
```

Example 31. `<prov:PMEDownload>` Request Message

5.3.1.11. Step 11: ProvS responds to the PMM with the PME.

The ProvS returns a portion of the PME. This sequence can be repeated multiple times as the PMM pulls sections of the PME rather than downloading the entire PME in a single call. This response would look something like:

```
<prov:PMEDownloadResponse remaining="83276" nextOffset="102400">
  <lu:Status code="OK"/>
  <prov:EngineData>
   ... base64 encoded data (100K worth) ...
  </prov:EngineData>
</prov:PMEDownloadResponse>
```

Example 32. `<prov:PMEDownloadResponse>` Message

5.3.1.12. Step 12: PMM Initialized the PM.

The PMM does what magic it must do to instantiate the PM within the trusted container. This may involve sending a message to a PME Init interface, or it may involve just starting up the PME executable (passing in the PMInitData and/or PMRTData as parameters).

This message and the method chosen by the PM is out-of-scope for Liberty and hence there is no Liberty ID-WSF defined interface to accomplish this task.

5.3.1.13. Step 13: PM Init routine returns OK.

The PM acknowledges receipt of the initialization parameters. This is another out-of-scope, non-Liberty message and may not even take place in some environments.

5.3.1.14. Step 14: PM sets its current status to Active.

The PM, upon completion of its initialization and verification of the passed in PMInitData and PMRTData (if any), uses the `<pmm:PMSetStatus>` interface to set its current status to *urn:liberty:prov:2007-09:status:Active*>.

Some implementations might use internal proprietary messages to accomplish this task (that would tightly bind the PM to a particular instance of a PMM) and others will use the Liberty defined message. That is an implementation decision.

In the case of using the Liberty defined message, the PM knows where to send this information and the PMID associated with the PM through some out-of-scope means (such as it being passed as an input parameter during the initialization stage).

This request would look something like:

```
<pmm:PMSetStatus>
  <prov:PMStatus>
    <prov:PMID issuer="http://provs-r-us.com">uuid:778349-283920-88379-5448739</prov:PMID>
    <prov:State>urn:liberty:prov:2007-09:status:Activated</prov:State>
  </prov:PMStatus>
</pmm:PMSetStatus>
```

Example 33. `<pmm:PMSetStatus>` Request Message

5.3.1.15. Step 15: PMM updates the PMD status at the ProvS.

The PMM uses the `<prov:PMSetStatus>` interface to update the status of the PMD at the ProvS.

This request would look something like:

```
<prov:PMSetStatus>
  <prov:PMStatus>
    <prov:PMID issuer="http://provs-r-us.com">uuid:239032-230328-92379-2397923</prov:PMID>
    <prov:State>urn:liberty:prov:2007-09:status:Active</prov:State>
  </prov:PMStatus>
</prov:PMSetStatus>
```

Example 34. `<prov:PMSetStatus>` Request Message

5.3.1.16. Step 16: ProvS responds to the PMM with an OK response.

The ProvS responds with an OK.

This message would look something like:

```
<prov:PMSetStatusResponse>
  <lu:Status code="OK" />
</prov:PMSetStatusResponse>
```

Example 35. <prov:PMDSetStatusResponse> Message

5.3.1.17. Step 17: PMM responds to status change with OK.

The PMM responds to the status update made in step 14 (Section 5.3.1.14) with an OK.

In this particular sequence the OK response is made after the PMM has updated the ProvS status (Section 5.3.1.15); however, there is no normative requirement for this particular ordering. The PMM could have sent this OK response immediately after the <pmm:PMSetStatus> request and then went on to update the status at the ProvS.

In any case, this response would look something like:

```
<pmm:PMSetStatusResponse>
  <lu:Status code="OK" />
</pmm:PMSetStatusResponse>
```

Example 36. <pmm:PMSetStatusResponse> Message

5.3.1.18. Step 18: PMM responds to RegApp with an OK.

The PMM responds to the provision request in step 5 (Section 5.3.1.5) with an OK.

The timing of this response is somewhat arbitrary. However, it should not take place until the PMM is satisfied that the PM is instantiated and available (at which point the RegApp could make use of it). So, this message could have been sent anytime after receipt of the status update in step 14 (Section 5.3.1.14).

This response would look something like:

```
<pmm:ProvisionResponse>
  <lu:Status code="OK" />
</pmm:ProvisionResponse>
```

Example 37. <pmm:ProvisionResponse> Message

5.3.2. Update Provisioning

This use case walks through a potential sequence of steps to update an existing PM (previously provisioned) on a client platform. In this case, the PMM is not directly addressable by the ProvS and as such, the ProvS must wait for the PMM to poll it for any update request.

The RegS is the controlling entity in this use case, building and issuing the update request through the ProvS. The PMID MUST be the same PMID that was used when the PM was originally provisioned (so that the ProvS will know which PMM to send the request to and so that the PMM will know which PM is being updated).

The sequence diagram below shows the sequence of steps between the various components:

1122

1123 **Figure 8. Update Workflow**

1124 ### 5.3.2.1. Step 1: The RegS requests an update at the ProvS.

1125 The RegS uses the `<prov:PMUpdate>` interface on the ProvS to register an update to a previously provisioned PM.
1126 This message would look something like:

```
<prov:PMUpdate>
  <prov:PMUpdateItem itemID="1" type="urn:liberty:prov:2007-09:ut:engine">
    <prov:PMDescriptor xs:id="2323923900239" >
      <prov:PMID issuer="http://provs-r-us.com">uuid:778349-283920-88379-5448739</prov:PMID>
      <prov:PMEngineRef>https://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
      <ds:Signature>
        ... signature data goes here ...
      </ds:Signature>
    </prov:PMDescriptor>
  </prov:PMUpdateItem>
  <dp:NotifyTo>
    <wsa:Address>https://provider.com/notifications</wsa:Address>
    <wsa:Metadata>
      <ds:ProviderID>http://provider.com/</ds:ProviderID>
      <ds:ServiceType>urn:liberty:dp:2007-09:notification</ds:ServiceType>
      <ds:Framework version="2.0" />
      <ds:SecurityContext>
        <ds:SecurityMechID>urn:liberty:security:2005-02:TLS:null</ds:SecurityMechID>
      </ds:SecurityContext>
    </wsa:Metadata>
  </dp:NotifyTo>
</prov:PMUpdate>
```

Example 38. `<prov:PMUpdate>` Request Message

5.3.2.2. Step 2: ProvS responds to RegS with WillNotify.

The ProvS acknowledges receipt of the update request. The ProvS must verify that it already has the correct PME for the updated PMD prior to acknowledging the receipt (or, if ProvS does not have the correct PME, it should return an error).

The RegS can use the `<prov:PMEUpload>` call to send the PME to the ProvS, if necessary.

The ProvS acknowledgement will look something like:

```
<prov:PMUpdateResponse>
  <lu:Status code="WillNotify"/>
</prov:PMUpdateResponse>
```

Example 39. `<prov:PMDUpdateResponse>` Message

5.3.2.3. Step 3: The PMM polls the ProvS for any requests.

The PMM in this case, is unable to expose a network accessible interface for any incoming requests and so must periodically poll the ProvS for any outstanding requests using the `<prov:Poll>` request. The frequency of these requests is under control of the ProvS and the PMM.

In this particular case, the PMM just happened to poll the ProvS shortly after the ProvS received the update request. In other cases the PMM may already have a waiting poll request (i.e., this message may have sent to the ProvS before the update request from the RegS) or the poll request could take place much later, depending upon the polling cycle.

This request would look something like:

```
<prov:Poll wait="300">
  <wsa:Action>urn:liberty:pmm:2007-09:PMUpdate</wsa:Action>
  <wsa:Action>urn:liberty:pmm:2007-09:PMDelete</wsa:Action>
  <wsa:Action>urn:liberty:pmm:2007-09:PMGetStatus</wsa:Action>
</prov:Poll>
```

Example 40. `<prov:Poll>` Request Message

5.3.2.4. Step 4: ProvS responds to the PMM with PMUpdate request.

The ProvS responds to the `<prov:Poll>` request with an OK status and includes the `<pmm:PMUpdate>` request for the PMM asking to be notified of the completion of the update at the same endpoint where the polling requests are delivered.

This response would look something like:

```
<prov:PollResponse>
  <lu:Status code="OK" />
  <dp:Request itemID="1">
    <pmm:PMUpdate>
      <pmm:PMUpdateItem itemID="1" type="urn:liberty:prov:2007-09:ut:engine">
        <prov:PMDescriptor xs:id="2323923900239" >
          <prov:PMID issuer="http://provs-r-us.com">uuid:778349-283920-88379-5448739</prov:PMID>
          <prov:PMEngineRef>https://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
          <ds:Signature>
            ... signature data goes here ...
          </ds:Signature>
        </prov:PMDescriptor>
      </pmm:PMUpdateItem>
      <dp:NotifyTo>
        <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
      </dp:NotifyTo>

    </pmm:PMUpdate>
  </dp:Request>
</prov:PollResponse>
```

Example 41. `<prov:PollResponse>` Message

5.3.2.5. Step 5: The PMM polls the ProvS again to send PMUpdateResponse.

The PMM immediately polls the ProvS again, this time including the response to the `<pmm:PMUpdate>` request that was included in the `<prov:PollResponse>` in step 4 (Section 5.3.2.4) above.

This poll, while accepting new requests, also tells the ProvS to not wait for new work, but return immediately, even if there are no new requests.

This request would look something like:

```
<prov:Poll wait="300">
  <wsa:Action>urn:liberty:pmm:2007-09:PMUpdate</wsa:Action>
  <wsa:Action>urn:liberty:pmm:2007-09:PMDelete</wsa:Action>
  <wsa:Action>urn:liberty:pmm:2007-09:PMGetStatus</wsa:Action>
  <dp:Response ref="1">
    <prov:UpdatePMResponse>
      <lu:Status code="WillNotify" />
    </prov:UpdatePMResponse>
  </dp:Response>
</prov:Poll>
```

Example 42. `<prov:Poll>` Request Message

5.3.2.6. Step 6: ProvS responds to the PMM with OK.

The ProvS responds to the `<prov:Poll>` request with an OK status and since there is no other outstanding request, nothing else. The ProvS also tells the PMM to poll again in 10 minutes. This response would look something like:

```
<prov:PollResponse nextPoll="600">
  <lu:Status code="OK" />
</prov:PollResponse>
```

Example 43. `<prov:PollResponse>` Message

5.3.2.7. Step 7: PMM retrieves PME information from the ProvS.

The PMM parses the PMD and noting that it does not have an instance of that PME invokes the `<prov:PMEGetInfo>` interface to obtain the PME information (size, hash, etc).

This request would **not** be necessary if the PMM already had an instance of the PME available locally (perhaps because another PM was instantiated which used the same PME, or because that particular PME was built into the platform).

This request would look something like:

```
<prov:PMEGetInfo>
  <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
</prov:PMEGetInfo>
```

Example 44. `<prov:PMEGetInfo>` Request Message

5.3.2.8. Step 8: ProvS responds to the PMM with the PME Info.

The ProvS returns the PME Information to the PMM. This response would look something like:

```
<prov:PMEGetInfoResponse>
  <lu:Status code="OK" />
  <prov:PMEInfo>
    <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
    <prov:PMECreatorID>http://reg.providers.com"</prov:PMECreatorID>
    <prov:PMEWhenCreated>2007-01-18T17:32:14Z</prov:PMEWhenCreated>
    <prov:PMEEnabled>true</prov:PMEEnabled>
    <prov:PMEWhenEnabled>2007-01-18T18:15:22Z</prov:PMEWhenEnabled>
    <prov:PMESize>185676</prov:PMESize>
    <prov:PMEHash method="...SHA-1">...SHA1 hash data ...</prov:PMEHash>
  </prov:PMEInfo>
</prov:PMEGetInfoResponse>
```

Example 45. `<prov:PMEGetInfoResponse>` Message

5.3.2.9. Step 9: PMM retrieves the updated PME from the ProvS.

The PMM initiates the download of the PME from the ProvS using the `<prov:PMEDownload>` interface.

This PMM specifies how much of the PME it wants to get in each `<prov:PMEDownload>` invocation and will usually invoke this several times to download the entire executable.

This request would look something like:

```
<prov:PMEDownload count="102400">
  <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
</prov:PMEDownload>
```

Example 46. `<prov:PMEDownload>` Request Message

5.3.2.10. Step 10: ProvS responds to the PMM with the PME.

The ProvS returns a portion of the PME. This sequence can be repeated multiple times as the PMM pulls sections of the PME rather than downloading the entire PME in a single call. This response would look something like:

```
<prov:PMEDownloadResponse remaining="83276" nextOffset="102400">
  <lu:Status code="OK"/>
  <prov:EngineData>
   ... base64 encoded data (100K worth) ...
  </prov:EngineData>
</prov:PMEDownloadResponse>
```

Example 47. `<prov:PMEDownloadResponse>` Message

5.3.2.11. Step 11: PMM updates the PM.

The PMM updates the PM. The actual mechanisms used to perform the update are out-of-scope for Liberty and therefore no messages or explicit steps are shown.

5.3.2.12. Step 12: PMM Notifies ProvS about update completion.

The PMM, upon completion of the update of the PM, uses the `<dp:Notification>` interface to notify the ProvS of the completion status of the update request (in this case a success).

This request would look something like:

```
<dp:Notification ref="...messageID-of-request...">
  <pmm:PMUpdateResponse>
    <lu:Status code="OK" />
  </pmm:PMUpdateResponse>
</dp:Notification>
```

Example 48. `<dp:Notification>` Request Message

5.3.2.13. Step 13: ProvS responds to notification with OK.

The ProvS responds to the notification with an OK.

In this particular sequence the OK response is made before the ProvS has sent the completion status notification to the original requester. However, there is no normative requirement for this particular ordering. The ProvS could have sent this OK response after it had sent the subsequent notification to the original requester.

In any case, this response would look something like:

```
<dp:NotificationResponse>
  <lu:Status code="OK" />
</dp:NotificationResponse>
```

Example 49. `<dp:NotificationResponse>` Message

5.3.2.14. Step 14: ProvS Notifies RegS about update completion.

The ProvS, upon receipt of the update notification from the PMM, sends its own notification to the RegS with the completion status of the request (in this case a success).

This request would look something like:

```
<dp:Notification ref="...messageID-of-request...">
  <prov:PMUpdateResponse>
    <lu:Status code="OK"/>
  </prov:PMUpdateResponse>
</dp:Notification>
```

Example 50. `<dp:Notification>` Request Message

5.3.2.15. Step 15: RegS responds to notification with OK.

The RegS responds to the notification with an OK.

This response would look something like:

```
<dp:NotificationResponse>
  <lu:Status code="OK" />
</dp:NotificationResponse>
```

Example 51. `<dp:NotificationResponse>` Message

References

Normative

[LibertyIdP] Cahill, Conor P., eds. "Liberty ID-WSF IDP Service Specification," Version 1.0, Liberty Alliance Project (14 December 2007). *http://www.projectliberty.org/specs*

[LibertySHPS] Cahill, Conor, eds. "Liberty ID-WSF Service Hosting and Proxying Service Specification," Version 1.0, Liberty Alliance Project (14 December 2007). *http://www.projectliberty.org/specs*

[LibertyPROV] Cahill, Conor P., eds. "Liberty ID-WSF Provisioning Service Specification," Version 1.0, Liberty Alliance Project (14 December 2007). *http://www.projectliberty.org/specs*

[LibertyPMM] Cahill, Conor P., eds. "Liberty ID-WSF Provisioned Module Manager Service Specification," Version 1.0, Liberty Alliance Project (14 December 2007). *http://www.projectliberty.org/specs*

[LibertyDisco] Cahill, Conor, Hodges, Jeff, eds. "Liberty ID-WSF Discovery Service Specification," Version 2.0-errata-v1.0, Liberty Alliance Project (29 November, 2006). *http://www.projectliberty.org/specs*

[LibertyIDPP] Kellomäki, Sampo, Lockhart, Rob, eds. "Liberty ID-SIS Personal Profile Service Specification," Version 1.1, Liberty Alliance Project (29 September, 2005). *http://www.projectliberty.org/specs*

[LibertyInteract] Aarts, Robert, Madsen, Paul, eds. "Liberty ID-WSF Interaction Service Specification," Version 2.0-errata-v1.0, Liberty Alliance Project (21 April, 2007). *http://www.projectliberty.org/specs*

[LibertyProtSchema] Cantor, Scott, Kemp, John, eds. "Liberty ID-FF Protocols and Schema Specification," Version 1.2-errata-v3.0, Liberty Alliance Project (14 December 2004). *http://www.projectliberty.org/specs*

[LibertyReg] Kemp, John, eds. "Liberty Enumeration Registry Governance," Version 1.1, Liberty Alliance Project (14 December, 2004). *http://www.projectliberty.org/specs*

[LibertySOAPAuthn] Hodges, Jeff, Aarts, Robert, Madsen, Paul, Cantor, Scott, eds. " Liberty ID-WSF Authentication, Single Sign-On, and Identity Mapping Services Specification ," v2.0-errata-1.0-01, Liberty Alliance Project (28 November, 2006). *http://www.projectliberty.org/specs*

[LibertySOAPBinding] Hodges, Jeff, Kemp, John, Aarts, Robert, Whitehead, Greg, Madsen, Paul, eds. "Liberty ID-WSF SOAP Binding Specification," Version 2.0-errata-v1.0, Liberty Alliance Project (21 April, 2007). *http://www.projectliberty.org/specs*

[LibertyPAOS] Aarts, Robert, Kemp, John, eds. "Liberty Reverse HTTP Binding for SOAP Specification," Version 2.0, Liberty Alliance Project (30 July, 2006). *http://www.projectliberty.org/specs*

[RFC2119] S. Bradner "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119, The Internet Engineering Task Force (March 1997). *http://www.ietf.org/rfc/rfc2119.txt*

[RFC2251] "Lightweight Directory Access Protocol (v3)," M. Wahl T. Howes S. Kille (December 1997). RFC 2251, Internet Engineering Task Force *http://www.ietf.org/rfc/rfc2251.txt*

[RFC2252] "Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions," M. Wahl A. Coulbeck T. Howes S. Kille (December 1997). RFC 2252, Internet Engineering Task Force *http://www.ietf.org/rfc/rfc2252.txt*

[RFC2891] Howes, T., Wahl, M., eds. (August 2000). "LDAP Control Extension for Server Side Sorting of Search Results," RFC 2891, Internet Engineering Task Force *http://www.ietf.org/rfc/rfc2891.txt*

[SAMLCore11] Maler, Eve, Mishra, Prateek, Philpott, Rob, eds. (2 September 2003). "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V1.1," SAML v1.1, OASIS Standard, Organization for the Advancement of Structured Information Standards *http://www.oasis-open.org/committees/download.php/3406/oasis-sstc-saml-core-1.1.pdf*

[SAMLCore2] Cantor, Scott, Kemp, John, Philpott, Rob, Maler, Eve, eds. (15 March 2005). "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0," SAML V2.0, OASIS Standard, Organization for the Advancement of Structured Information Standards *http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf*

[SAMLProf2] Hughes, John, Cantor, Scott, Hodges, Jeff, Hirsch, Frederick, Mishra, Prateek, Philpott, Rob, Maler, Eve, eds. (15 March, 2005). "Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0," SAML V2.0, OASIS Standard, Organization for the Advancement of Structured Information Standards *http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf*

[Schema1-2] Thompson, Henry S., Beech, David, Maloney, Murray, Mendelsohn, Noah, eds. (28 October 2004). "XML Schema Part 1: Structures Second Edition," Recommendation, World Wide Web Consortium *http://www.w3.org/TR/xmlschema-1/*

[SOAPv1.1] "Simple Object Access Protocol (SOAP) 1.1," Box, Don, Ehnebuske, David , Kakivaya, Gopal, Layman, Andrew, Mendelsohn, Noah, Nielsen, Henrik Frystyk, Winer, Dave, eds. World Wide Web Consortium W3C Note (08 May 2000). *http://www.w3.org/TR/2000/NOTE-SOAP-20000508/*

[XML] Bray, Tim, Paoli, Jean, Sperberg-McQueen, C. M., Maler, Eve, Yergeau, Francois, eds. (04 February 2004). "Extensible Markup Language (XML) 1.0 (Third Edition)," Recommendation, World Wide Web Consortium *http://www.w3.org/TR/2004/REC-xml-20040204*

[XMLDsig] Eastlake, Donald, Reagle, Joseph, Solo, David, eds. (12 Feb 2002). "XML-Signature Syntax and Processing," Recommendation, World Wide Web Consortium *http://www.w3.org/TR/xmldsig-core*

[XPATH] Clark , J., DeRose , S., eds. (16 November 1999). "XML Path Language (XPath) Version 1.0 ," Recommendation, W3C *http://www.w3.org/TR/xpath [August 2003].*

Informative

[LibertyIDWSFGuide] Weitzel, David, eds. "Liberty ID-WSF Implementation Guide," Version 2.0-02, Liberty Alliance Project (13 January, 2005). *http://www.projectliberty.org/specs*

[LibertyIDPPGuide] Kellomäki, Sampo, Lockhart, Rob, eds. "Liberty ID-SIS Personal Profile Service Implementation Guidelines," Version 1.1, Liberty Alliance Project (29 September, 2005). *http://www.projectliberty.org/specs*

[LibertyIDWSFOverview] Tourzan, Jonathan, Koga, Yuzo, eds. "Liberty ID-WSF Web Services Framework Overview," Version 2.0, Liberty Alliance Project (30 July, 2006). *http://www.projectliberty.org/specs*