



# 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

1 **Notice**

2 This document has been prepared by Sponsors of the Liberty Alliance. Permission is hereby granted to use the  
3 document solely for the purpose of implementing the Specification. No rights are granted to prepare derivative works  
4 of this Specification. Entities seeking permission to reproduce portions of this document for other uses must contact  
5 the Liberty Alliance to determine whether an appropriate license for such use is available.

6 Implementation of certain elements of this document may require licenses under third party intellectual property  
7 rights, including without limitation, patent rights. The Sponsors of and any other contributors to the Specification are  
8 not and shall not be held responsible in any manner for identifying or failing to identify any or all such third party  
9 intellectual property rights. **This Specification is provided "AS IS," and no participant in the Liberty Alliance**  
10 **makes any warranty of any kind, express or implied, including any implied warranties of merchantability,**  
11 **non-infringement of third party intellectual property rights, and fitness for a particular purpose.** Implementers  
12 of this Specification are advised to review the Liberty Alliance Project's website (<http://www.projectliberty.org/>) for  
13 information concerning any Necessary Claims Disclosure Notices that have been received by the Liberty Alliance  
14 Management Board.

15 Copyright © 2007 Adobe Systems; Agencia Catalana De Certificacio; America Online, Inc.; American Express  
16 Company; Amsoft Systems Pvt Ltd.; Avatier Corporation; BIPAC; BMC Software, Inc.; Bank of America  
17 Corporation; Beta Systems Software AG; British Telecommunications plc; Computer Associates International, Inc.;  
18 Credentica; Dan Combs; Danish National IT and Telecom Agency; DataPower Technology, Inc.; Deutsche Telekom  
19 AG, T-Com; Diamelle Technologies, Inc.; Diversinet Corp.; Drummond Group Inc.; Enosis Group LLC; Entrust,  
20 Inc.; Epok, Inc.; Ericsson; Falkin Systems LLC; Fidelity Investments; Forum Systems, Inc.; France Télécom; French  
21 Government Agence pour le développement de l'administration électronique (ADAE); Fugen Solutions, Inc; Fulvens  
22 Ltd.; GSA Office of Governmentwide Policy; Gamefederation; Gemalto; General Motors; GeoFederation; Giesecke  
23 & Devrient GmbH; Guy Huntington; Hewlett-Packard Company; Hochhauser & Co., LLC; IBM Corporation; Intel  
24 Corporation; Intuit Inc.; Kantega; Kayak Interactive; Livo Technologies; Luminance Consulting Services; Mark  
25 Wahl; Mary Ruddy; MasterCard International; MedCommons Inc.; Mobile Telephone Networks (Pty) Ltd; Mortgage  
26 Bankers Association (MBA); NEC Corporation; NTT DoCoMo, Inc.; Netegrity, Inc.; NHK (Japan Broadcasting  
27 Corporation) Science & Technical Research Laboratories; Neustar, Inc.; New Zealand Government State Services  
28 Commission; Nippon Telegraph and Telephone Corporation; Nokia Corporation; Novell, Inc.; OpenNetwork; Oracle  
29 Corporation; Ping Identity Corporation; Postsecondary Electronic Standards Council (PESC); RSA Security Inc.;  
30 Reach; Reactivity Inc.; Rob Marano; Royal Mail Group plc; SAP AG; SanDisk Corporation; Senforce; Sharp  
31 Laboratories of America; Sigaba; SmartTrust; Sony Corporation; Sun Microsystems, Inc.; Supremacy Financial  
32 Corporation; Symlabs, Inc.; Telecom Italia S.p.A.; Telefónica Móviles, S.A.; Telenor R&D; Thales e-Security;  
33 Trusted Network Technologies; UNINETT AS; UTI; VeriSign, Inc.; Vodafone Group Plc.; Wave Systems Corp.  
34 Wells Fargo; All rights reserved.

35 Liberty Alliance Project  
36 Licensing Administrator  
37 c/o IEEE-ISTO  
38 445 Hoes Lane  
39 Piscataway, NJ 08855-1331, USA  
40 info@projectliberty.org

## 41 Contents

42	1. Introduction	4
43	1.1. Notation and Conventions	4
44	1.1.1. XML Namespaces	4
45	2. Basic Concepts	5
46	2.1. Actors	5
47	2.2. Transactions	6
48	2.3. Modes of Operation	7
49	2.4. Client Identification	7
50	3. Identity Provider Operations	9
51	3.1. SSO	9
52	3.1.1. Self-Asserted SSO	9
53	3.1.2. Facilitated SSO	9
54	3.1.3. Delegated SSO	9
55	3.1.4. Minting Assertions	10
56	3.1.5. Hoarding Credentials	11
57	3.1.6. Using Delegated SSO	12
58	3.2. Identity Federation	13
59	3.2.1. Self-Asserted Identity Federations	13
60	3.2.2. Online Federations	13
61	4. Client hosted services	14
62	4.1. Client Service Components	14
63	4.2. Services Data	15
64	4.3. SHPS Examples	15
65	4.3.1. Hosted Service Instance Registration	15
66	4.3.2. Proxied Service Instance Registration	19
67	4.3.3. Proxied Service Invocation	22
68	5. Provisioning	26
69	5.1. Provisioning Components	26
70	5.2. Provisioning Data	27
71	5.3. Provisioning Examples	27
72	5.3.1. Initial Provisioning	28
73	5.3.2. Update Provisioning	34
74	References	41

## 75 1. Introduction

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

### 78 1.1. Notation and Conventions

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

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

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

#### 86 1.1.1. XML Namespaces

87 The following XML namespaces are referred to in this document:

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

89 *urn:liberty:prov:2007-09*

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

91 *urn:liberty:pmm:2007-09*

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

93 *urn:liberty:shps:2007-09*

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

95 *urn:liberty:idp:2007-09*

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

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

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

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

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

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

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

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

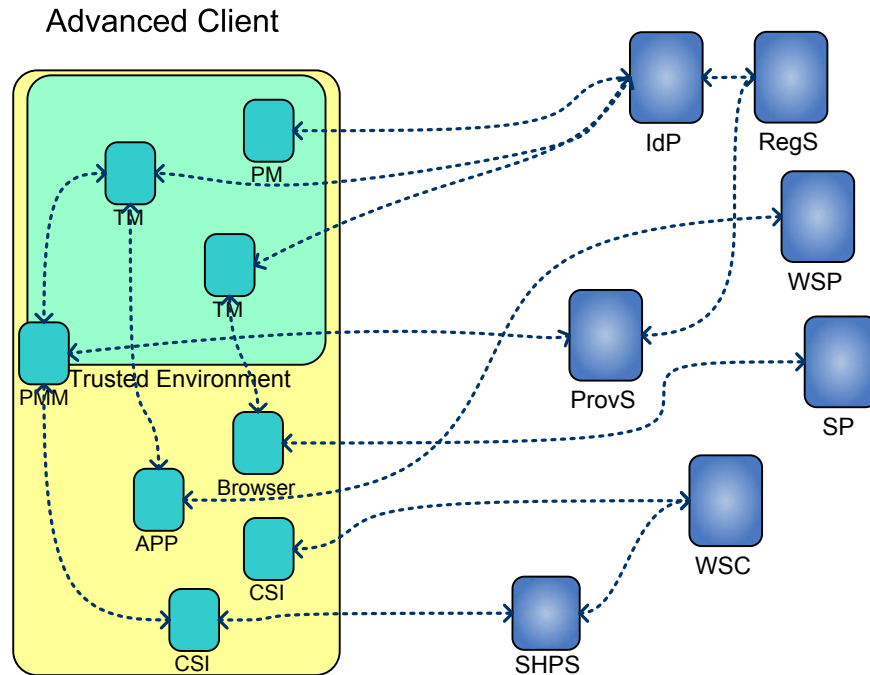
## 104 2. Basic Concepts

### 105 2.1. Actors

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

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

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



138

139

Figure 1. Actors

## 140 2.2. Transactions

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

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

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

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

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

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

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

- 160           • collect credentials from the principal and relay them to the IdP for validation.
- 161           • assert internally stored credentials to the IdP as an authentication for the principal (similar to what a SIM
- 162           card does within a mobile phone).
- 163           • locally authenticate the principal through various means (PIN, PW, biometrics, etc.) and then using an
- 164           internal credential, assert the completion of that authentication to the IdP.
- 165    • **Web Service Instance** - the client hosts one or CSIs. For privacy and/or availability reasons, the CSI may host or
- 166    proxy its service through a network based SHPS or it may expose the service directly.
- 167    • **Provisioning** - modules (TMs, CSIs) can be provisioned over the wire to an advanced client. This functionality
- 168    includes the full lifecycle support (provision, update, delete).

## 169 2.3. Modes of Operation

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

- 173    • **Connected** - the Advanced Client is fully connected to the network and generally all parties to a transaction could
- 174    communicate with each other if necessary. In this mode, the Advanced Client can choose to act as a simple
- 175    facilitator of the actual operation or for various reasons (such as privacy, load balancing, etc.) the Advanced Client
- 176    can take a more active role, providing delegated authentication and/or web services.
- 177    • **Disconnected** - the Advanced Client does not have connectivity to one or more parties in a transaction (such as
- 178    not having connectivity to the IdP during an authentication transaction). This mode limits the Advanced Client
- 179    ability to participate in facilitated operations and can restrict the availability of services exposed directly by the
- 180    Advanced Client.
- 181    Note that disconnected mode can also be used by the Advanced Client as a privacy preserving mode (where the
- 182    IdP doesn't know exactly when a Advanced Client SSOs the user to a relying party, nor how many times).

## 183 2.4. Client Identification

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

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

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

- 192    • **unique identifier** - an identifier that identifies that specific Advanced Client uniquely amongst all other Advanced
- 193    Clients (even on the same device).
- 194    Using a unique identifier for the ProviderID of a Advanced Client opens the principal's privacy to potential
- 195    correlation attacks and so it should not be used when privacy is of paramount importance.
- 196    However, using a unique identifier does increase the traceability of operations (which can be a good thing from a
- 197    security perspective) and it also allows an individual Advanced Client to be "canceled" without impacting other
- 198    Advanced Clients issued by the same or other parties.

- 199 • **shared identifier** - an identifier that is shared amongst a group of Advanced Clients (typically manufactured by  
200 the same party).
- 201 The definition of what makes up a "group" of Advanced Clients suitable for a shared identifier is up to the issuing  
202 party. A common solution is to use the same identifier across all instances of a Advanced Client issued by the  
203 same party with the same version of the software core for the Advanced Client (e.g., the same version of the  
204 Advanced Client).
- 205 If the group is large enough in a particular context, the use of a **shared identifier** retains some of the security  
206 benefits of the **unique identifier** while at the same time retaining some of the privacy benefits of the **common**  
207 **identifier**.
- 208 For example, if there's a problem with a particular version of the Advanced Client, it can be black-listed, requiring  
209 an upgrade to a fixed version prior to subsequent use), while also maintaining the privacy benefits of a common  
210 identifier as the correlation value is very small when there are many Advanced Clients with the same identifier.
- 211 • **common identifier** - a generic identifier assigned to all Advanced Clients that do not have a shared or unique  
212 identifier. Liberty has reserved the following URN for this case: *urn:liberty:idp:2007-09:ProviderID:Common*
- 213 The common identifier eliminates the ProviderID as a potential correlation factor (although it doesn't get rid of  
214 other correlation handles such as the Advanced Client's IP address), but it also means that other means must be  
215 enacted to deal with security issues like revocation.
- 216 The Advanced Client's ProviderID should be used with strong caution in making security decisions unless it is  
217 somehow cryptographically protected in the transaction by the issuing party. One such way to protect the ProviderID  
218 is to include that provider ID in any security tokens issued to the Advanced Client to show that the token is associated  
219 with that ProviderID.



## 220 **3. Identity Provider Operations**

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

### 223 **3.1. SSO**

#### 224 **3.1.1. Self-Asserted SSO**

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

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

232 In order to expose these services, the TM need only follow the off-the-shelf protocols (Liberty AS, Liberty DS, SAML  
233 2.0 Browser profile, etc.).

#### 234 **3.1.2. Facilitated SSO**

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

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

#### 238 **3.1.3. Delegated SSO**

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

##### 242 **3.1.3.1. Credentials for SSO**

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

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

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

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

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

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

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

### 264 3.1.4. Minting Assertions

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

267 • **minting assertion** (MING) – An assertion issued by an IdP authorizing the TM to mint MEDs (typically with  
268 conditions limiting the applicability of the MEDs). See [LibertyIdP] for a detailed description of MINGs.

269 • **ProviderID** of the relying party(ies) for which the TM would like to generate assertions.

270 • **nameID** for the principal at the relying party(ies).

#### 271 3.1.4.1. Obtaining Minting Credential

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

274 • Set the `assnType` attribute to:

275 `urn:liberty:idp:2007-09:purpose:minting`

276 • Include a `<saml2p:AuthnRequest>` for each provider for which a MING is desired.

277 • If correlation across providers is a privacy concern, specify a unique public key for each  
278 `<saml2p:AuthnRequest>` in the `<saml2p:Subject>`'s `<saml2p:SubjectConfirmation>` element.

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

281 The TM is normally expected to generate private/public key pairs as necessary for minting assertions. It is possible  
282 that some environments may utilize some proprietary means of key management such that the IdP knows which  
283 keys to use for which provider and specifying them on this request would not be necessary in such environments.

284 • Specify the ProviderIDs for the providers in the `<saml2:Conditions>`'s `<saml2:AudienceRestriction>`  
285 element. If assertions for multiple providers are desired **AND** correlation is **NOT** a concern, mul-  
286 tiple provider IDs can be specified within different `<saml2:Audience>` elements in the same  
287 `<saml2:AudienceRestriction>` element.

288 If correlation is a concern, the MING for each provider should be requested within separate  
289 `<saml2p:AuthnRequest>` elements in the `<idp:GetAssertion>` element.

290 • If desired, specify the requested expiration time for the MING(s) using the `<saml2:Conditions>`'s  
291 `NotOnOrAfter` attribute.

292 • Other desired conditions may be specified including some of the conditions we added above. For example, the  
293 `<AuthnContextRestriction>` by the TM to specify the AuthnContexts it plans to use.

294 A sample `<idp:GetAssertion>` message:

```
295 <idp:GetAssertion purpose="urn:liberty:idp:2007-09:purpose:minting">
296
297   <saml2p:AuthnRequest ID="ID_2343823023823" Version="2.0"
298     IssueInstant="2006-06-23T15:38:46Z">
299     <saml2:Subject>
300       <saml2:SubjectConfirmation Method="...:holder-of-key">
301         <saml2:SubjectConfirmationData>
302           <ds:KeyInfo>Key info for Minting Assn (TM Public Key)</ds:KeyInfo>
303         </saml2:SubjectConfirmationData>
304       </saml2:SubjectConfirmation>
305     </saml2:Subject>
306     <saml2p:NameIDPolicy Format="...:persistent"/>
307     <saml2:Conditions NotOnOrAfter="2006-07-23T15:38:46Z">
308       <saml2:AudienceRestriction>
309         <saml2:Audience>Provider 2</saml2:Audience>
310       </saml2:AudienceRestriction>
311     </saml2:Conditions>
312   </saml2p:AuthnRequest>
313 </idp:GetAssertion>
314
315
```

316 **Example 1. Example `<GetAssertion>` Message**

### 317 **3.1.4.2. Minting a Credential**

318 The process of minting a credential follows the required processing in the SAML 2.0 specifications. The only added  
319 element here is that the `<saml2:Advice>` element MUST contain the MING. All other processing is pure SAML  
320 2.0.

### 321 **3.1.5. Hoarding Credentials**

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

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

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

#### 330 **3.1.5.1. Obtaining Credentials**

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

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

334 • Set the `assnType` attribute to:

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

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

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

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

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

346 • Other desired conditions, `AuthnContexts`, etc., may be specified.

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

```
348 <idp:GetAssertion purpose="urn:liberty:idp:2007-09:purpose:sso">
349
350   <saml2p:AuthnRequest ID="ID_153282378" Version="2.0"
351     IssueInstant="2006-06-23T15:38:46Z">
352     <saml2p:NameIDPolicy Format="...:persistent"/>
353     <saml2:Conditions NotOnOrAfter="2006-07-23T15:38:46Z">
354       <saml2:AudienceRestriction>
355         <saml2:Audience>Provider 2</saml2:Audience>
356       </saml2:AudienceRestriction>
357     </saml2:Conditions>
358   </saml2p:AuthnRequest>
359
360 </idp:GetAssertion>
361
```

362 **Example 2. Example `<GetAssertion>` Message**

363 And an example response to that request:

```
364 <idp:GetAssertionResponse>
365   <lu:Status code="OK"/>
366   <idp:GetAssertionResponseItem ref="153282378">
367     <idp:AssertionItem created="false">
368       <saml2:Assertion>..Provider 2 Assertion.. </saml2:Assertion>
369     </idp:AssertionItem>
370   </idp:GetAssertionResponseItem>
371 </idp:GetAssertionResponse>
372
```

373 **Example 3. Example `<GetAssertionResponse>` Message**

### 374 3.1.6. Using Delegated SSO

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

#### 378 3.1.6.1. Browser Based SSO

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

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

382 • expose a SAML 2.0 Enhanced Client/Proxy (ECP) to relying parties (insert the necessary headers and process the  
383 incoming requests)

384 • communicate with the TM to obtain the necessary credential(s) to answer the incoming requests (treat the TM as  
385 the IdP).

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

### 388 3.1.6.2. SSO to WSPs

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

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

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

## 397 3.2. Identity Federation

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

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

### 404 3.2.1. Self-Asserted Identity Federations

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

### 407 3.2.2. Online Federations

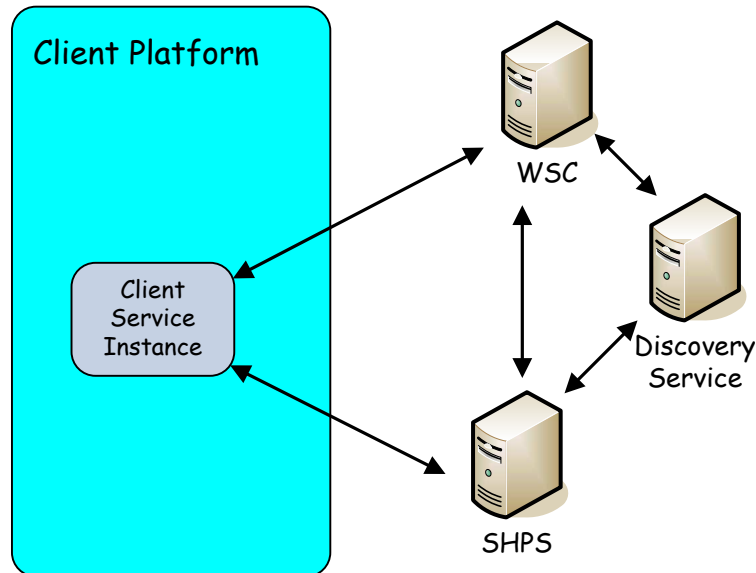
408 When the TM is able to connect to the IdP, we **strongly** recommend that the TM treat the federation event as a proxy  
409 event and use the IdP for the federation data (so that the IdP creates the federation handle at that time).

## 410 4. Client hosted services

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

### 414 4.1. Client Service Components

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



416

417

Figure 2. CSI Actors

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 464 4.2. Services Data

465 The following objects are defined by the client services system:

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

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

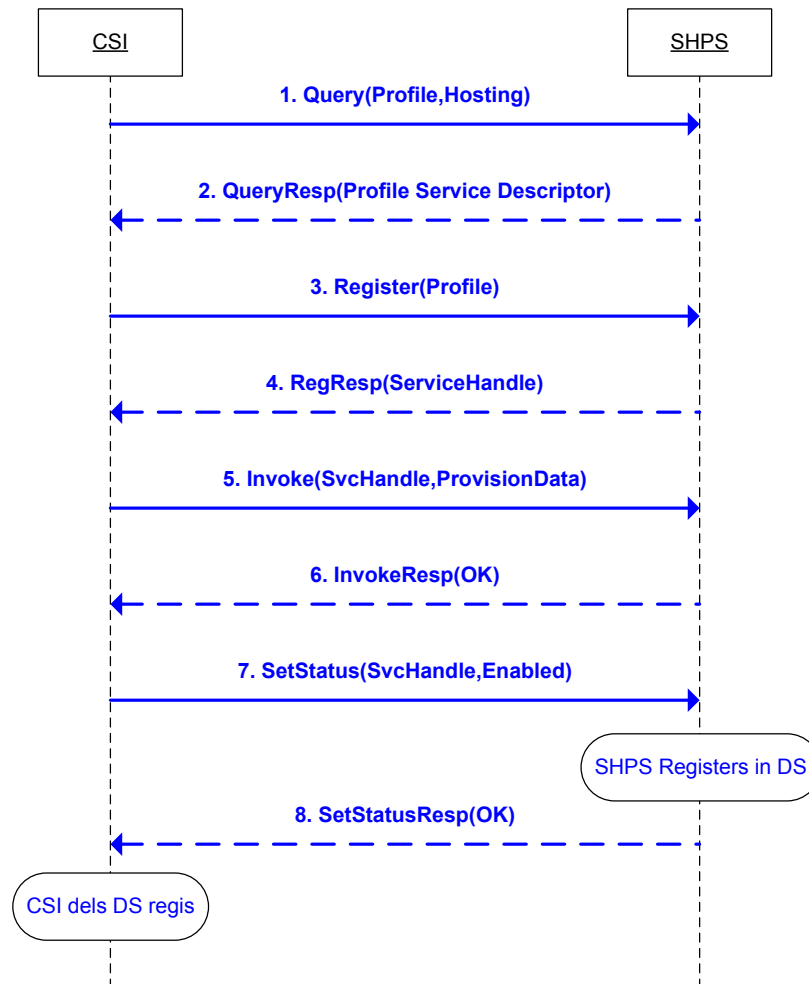
### 471 4.3. SHPS Examples

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

#### 474 4.3.1. Hosted Service Instance Registration

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

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



478

479 **Figure 3. Hosted Service Instance Registration Workflow**

480 Things to note about the diagram:

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



#### 486 4.3.1.1. Step 1: Query for supported service info

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

```
488 <shps:Query>
489   <disco:RequestedService>
490     <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
491     <disco:SecurityMechID>urn:liberty:sm:2006-08:TLS:SAMLV2</disco:SecurityMechID>
492     <disco:Framework version="2.0" />
493     <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:hosted</shps:ServiceMode>
494   </disco:RequestedService>
495 </shps:Query>
496
```

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

#### 498 4.3.1.2. Step 2: SHPS responds with service info

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

```
501 <shps:QueryResponse>
502   <lu:Status code="OK" />
503   <wsa:EndpointReference>
504     <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
505     <wsa:Metadata>
506       <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:hosted</shps:ServiceMode>
507       <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
508       <disco:Framework version="2.0" />
509       <disco:SecurityContext>
510         <disco:SecurityMechID>urn:liberty:security:2005-02:TLS:SAML</disco:SecurityMechID>
511       </disco:SecurityContext>
512     </wsa:Metadata>
513   </wsa:EndpointReference>
514 </shps:QueryResponse>
515
```

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

#### 517 4.3.1.3. Step 3: Register Hosted Service Instance

518 The CSI submits a registration request for the profile service.

```
519 <shps:Register>
520   <wsa:EndpointReference lu:itemID="1">
521     <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
522     <wsa:Metadata>
523       <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:hosted</shps:ServiceMode>
524       <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
525       <disco:Framework version="2.0" />
526       <disco:SecurityContext>
527         <disco:SecurityMechID>urn:liberty:security:2006-08:TLS:SAMLV2</disco:SecurityMechID>
528       </disco:SecurityContext>
529     </wsa:Metadata>
530   </wsa:EndpointReference>
531 </shps:Register>
532
```

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

#### 534 4.3.1.4. Step 4: Registration response from SHPS

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

```
536 <shps:RegisterResponse>
537   <lu:Status code="OK" />
538   <shps:RegisterResponseItem ref="1">
539     <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
540   </shps:RegisterResponseItem>
541 </shps:RegisterResponse>
542
```

543 **Example 7. <shps:RegisterResponse> Request Message**

#### 544 4.3.1.5. Step 5: Initialize data in hosted service

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

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

```
549 <shps:Invoke>
550   <shps:InvokeItem itemID="1">
551     <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
552     <pp:Modify>
553       <pp:ModifyItem>
554         ... modification data goes here ...
555       </pp:ModifyItem>
556     </pp:Modify>
557   </shps:InvokeItem>
558 </shps:Invoke>
559
```

560 **Example 8. <shps:Invoke> Request Message**

#### 561 4.3.1.6. Step 6: Initialization response

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

```
566 <shps:InvokeResponse>
567   <lu:Status code="OK" />
568   <shps:InvokeResponseItem ref="1">
569     <pp:ModifyResponse>
570       ... modification response data goes here ...
571     </pp:ModifyResponse>
572   </shps:InvokeResponseItem>
573 </shps:InvokeResponse>
574
```

575 **Example 9. <shps:InvokeResponse> Request Message**

#### 576 4.3.1.7. Step 7: Enable the service instance

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

```
580 <shps:SetStatus>
581   <shps:SetStatusItem itemID="1" >
582     <shps:ServiceStatus>urn:liberty:shps:2007-09:status:enabled</shps:ServiceStatus>
583     <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
584   </shps:SetStatusItem>
585 </shps:SetStatus>
586
```

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

### 588 4.3.1.8. Step 8: Service Enablement Response

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

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

```
593 <shps:SetStatusResponse>
594   <lu:Status code="OK" />
595 </shps:SetStatusResponse>
596
```

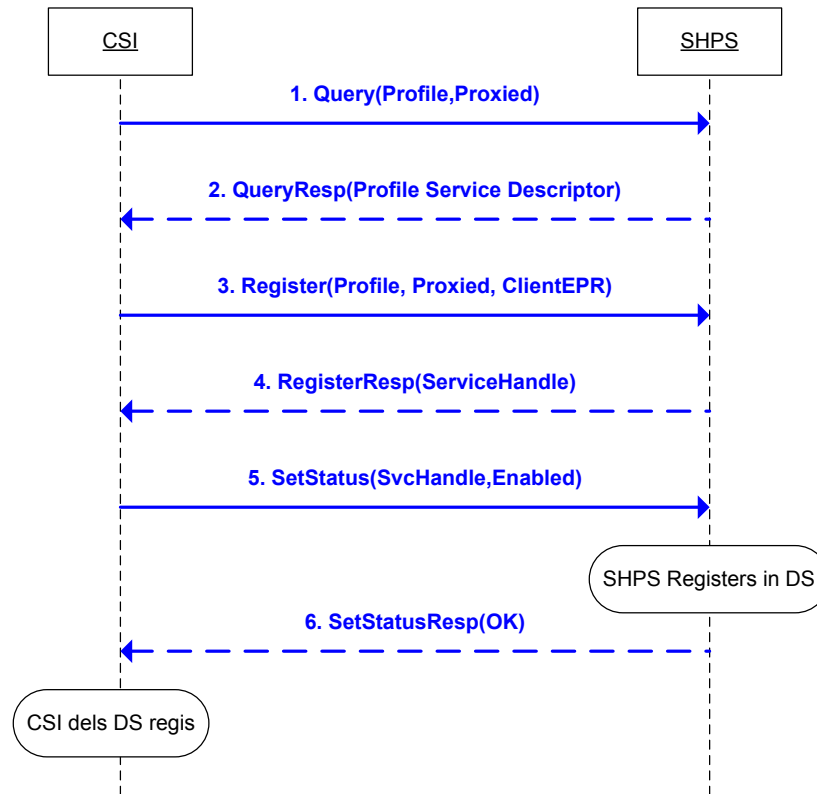
597 **Example 11. `<shps:SetStatusResponse>` Request Message**

### 598 4.3.2. Proxied Service Instance Registration

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

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

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



604

605

**Figure 4. Proxied Service Instance Registration Workflow**

606 Things to note about the diagram:

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

#### 612 4.3.2.1. Step 1: Query for supported service info

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

614 service.

```

615 <shps:Query>
616   <disco:RequestedService>
617     <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
618     <disco:SecurityMechID>urn:liberty:sm:2006-08:TLS:SAMLV2</disco:SecurityMechID>
619     <disco:Framework version="2.0" />
620     <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:proxied</shps:ServiceMode>
621   </disco:RequestedService>
622 </shps:Query>
623

```

624

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

#### 625 4.3.2.2. Step 2: SHPS responds with service info

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

```
628 <shps:QueryResponse>
629   <lu:Status code="OK" />
630   <wsa:EndpointReference>
631     <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
632     <wsa:Metadata>
633       <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:proxied</shps:ServiceMode>
634       <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
635       <disco:Framework version="2.0" />
636       <disco:SecurityContext>
637         <disco:SecurityMechID>urn:liberty:security:2005-02:TLS:SAML</disco:SecurityMechID>
638       </disco:SecurityContext>
639     </wsa:Metadata>
640   </wsa:EndpointReference>
641 </shps:QueryResponse>
642
```

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

#### 644 4.3.2.3. Step 3: Register Proxied Service Instance

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

```
647 <shps:Register>
648   <wsa:EndpointReference lu:itemID="1">
649     <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
650     <wsa:Metadata>
651       <shps:ServiceMode>urn:liberty:shps:2007-09:svcmode:proxied</shps:ServiceMode>
652       <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
653       <disco:Framework version="2.0" />
654       <disco:SecurityContext>
655         <disco:SecurityMechID>urn:liberty:security:2006-08:TLS:SAMLV2</disco:SecurityMechID>
656       </disco:SecurityContext>
657       <shps:CallbackEPR>
658         <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
659       </shps:CallbackEPR>
660     </wsa:Metadata>
661   </wsa:EndpointReference>
662 </shps:Register>
663
```

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

#### 665 4.3.2.4. Step 4: Registration response from SHPS

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

```
667 <shps:RegisterResponse>
668   <lu:Status code="OK" />
669   <shps:RegisterResponseItem ref="1">
670     <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
671   </shps:RegisterResponseItem>
672 </shps:RegisterResponse>
673
```

674 **Example 15. <shps:RegisterResponse> Request Message**

#### 675 4.3.2.5. Step 5: Enable the service instance

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

```
679 <shps:SetStatus>
680   <shps:SetStatusItem itemID="1" >
681     <shps:ServiceStatus>urn:liberty:shps:2007-09:status:enabled</shps:ServiceStatus>
682     <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
683   </shps:SetStatusItem>
684 </shps:SetStatus>
685
```

686 **Example 16. <shps:SetStatus> Request Message**

#### 687 4.3.2.6. Step 6: Service Enablement Response

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

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

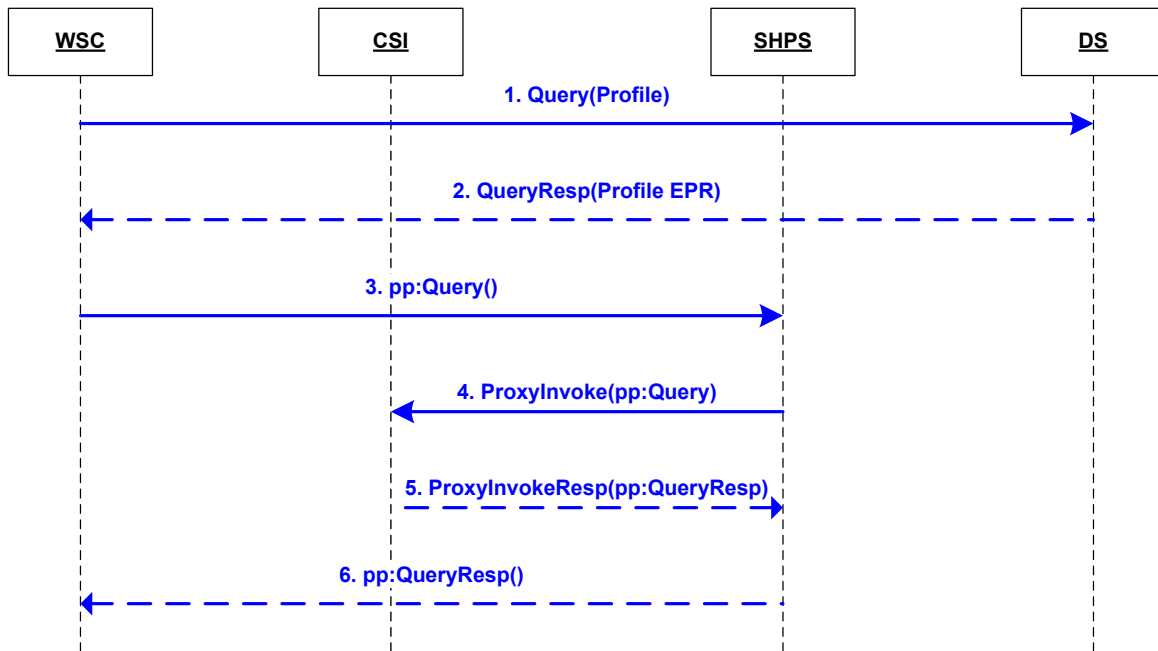
```
692 <shps:SetStatusResponse>
693   <lu:Status code="OK" />
694 </shps:SetStatusResponse>
695
```

696 **Example 17. <shps:SetStatusResponse> Request Message**

#### 697 4.3.3. Proxied Service Invocation

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

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



700

701

**Figure 5. Proxied Service Invocation Workflow**

702 Things to note about the diagram:

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

#### 708 4.3.3.1. Step 1: Discover the profile service provider

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

```

711 <disco:Query>
712   <disco:RequestedService>
713     <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
714
715     <disco:SecurityMechID>urn:liberty:security:2006-08:ClientTLS:SAMLV2</disco:SecurityMechID>
716     <disco:SecurityMechID>urn:liberty:security:2005-02:ClientTLS:SAML</disco:SecurityMechID>
717     <disco:SecurityMechID>urn:liberty:security:2006-08:TLS:SAMLV2</disco:SecurityMechID>
718     <disco:Framework version="2.0" />
719   </disco:RequestedService>
720 </disco:Query>
721

```

722

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

#### 723 4.3.3.2. Step 2: DS responds with service EPR

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

```
725 <disco:QueryResponse>
726   <lu:Status code="OK" />
727   <wsa:EndpointReference disco:notOnOrAfter="2007-12-08T13:23:40Z" >
728     <wsa:Address>http://shps.services.com/pp</wsa:Address>
729     <wsa:Metadata>
730       <disco:Abstract>Bob's personal profile</disco:Abstract>
731       <disco:ProviderID>http://services.com/</disco:ProviderID>
732       <disco:ServiceType>urn:liberty:id-sis-pp:2003-08</disco:ServiceType>
733       <disco:Framework version="2.0" />
734       <disco:SecurityContext>
735         <disco:SecurityMechID>urn:liberty:security:2005-02:TLS:SAMLV2</disco:SecurityMechID>
736         <sec:Token usage="urn:liberty:security:tokenusage:2006-08:SecurityToken">
737           <saml2:Assertion ...>
738             .... SAML Token data goes here ....
739           </saml2:Assertion>
740         </sec:Token>
741       </disco:SecurityContext>
742     </wsa:Metadata>
743   </wsa:EndpointReference>
744 </disco:QueryResponse>
745
```

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

#### 747 4.3.3.3. Step 3: WSC Invokes Profile service at SHPS

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

```
750 <pp:Query>
751   ... query data goes here ...
752 </pp:Query>
753
```

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

#### 755 4.3.3.4. Step 4: Proxied invocation of CSI from SHPS

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

```
757 <shps:ProxyInvoke>
758   <shps:ProxyInvokeItem itemID="1">
759     <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
760     <shps:InvocationContext>
761       <shps:InvokingProvider>http://services.corp.com</shps:InvokingProvider>
762       <disco:SecurityMechID>urn:liberty:security:2006-08:TLS:SAMLV2</disco:SecurityMechID>
763     </shps:InvocationContext>
764     <pp:Query>
765       ... query data goes here ...
766     </pp:Query>
767   </shps:ProxyInvokeItem>
768 </shps:ProxyInvoke>
769
```

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

#### 771 4.3.3.5. Step 5: Proxied Response from CSI to SHPS



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

```
773 <shps:ProxyInvokeResponse>
774   <lu:Status code="OK" />
775   <shps:ProxyInvokeResponseItem ref="1">
776     <shps:ServiceHandle>uuid:23023-023802-2032023-0238023</shps:ServiceHandle>
777     <shps:ResponseHeaders>
778       <sb:UsageDirectives> ..... </sb:UsageDirectives>
779     </shps:ResponseHeaders>
780     <pp:QueryResponse>
781       ... query response data goes here ...
782     </pp:QueryResponse>
783   </shps:ProxyInvokeResponseItem>
784 </shps:ProxyInvokeResponse>
785
```

786 **Example 22. `<shps:ProxyInvokeResponse>` Request Message**

#### 787 **4.3.3.6. Step 6: SHPS passes response to WSC**

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

```
789 <pp:QueryResponse>
790   ... query response data goes here ...
791 </pp:QueryResponse>
792
```

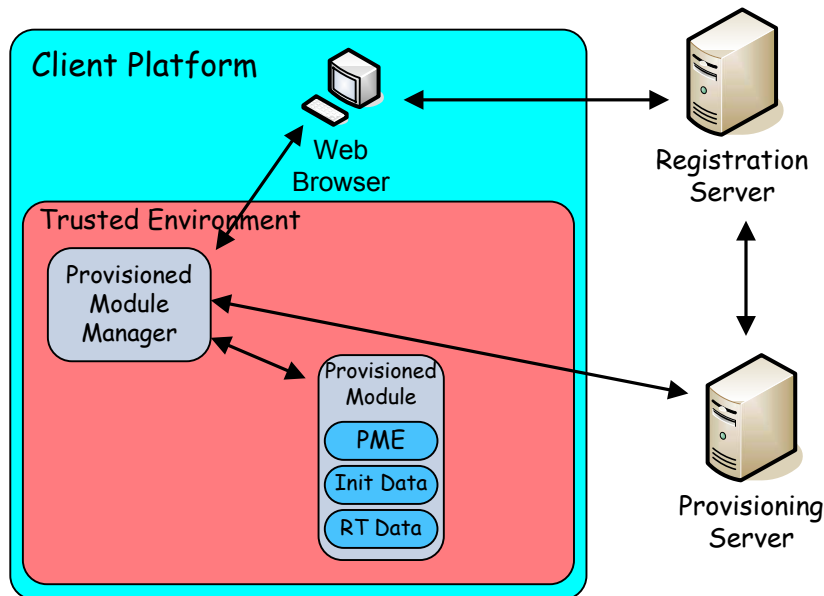
793 **Example 23. `<pp:QueryResponse>` Request Message**

## 794 5. Provisioning

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

### 801 5.1. Provisioning Components

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



803

804 **Figure 6. Provisioning Components**

805 Things to note about this diagram:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 853 5.2. Provisioning Data

854 The following objects are defined by the provisioning system:

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

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

### 860 5.3. Provisioning Examples

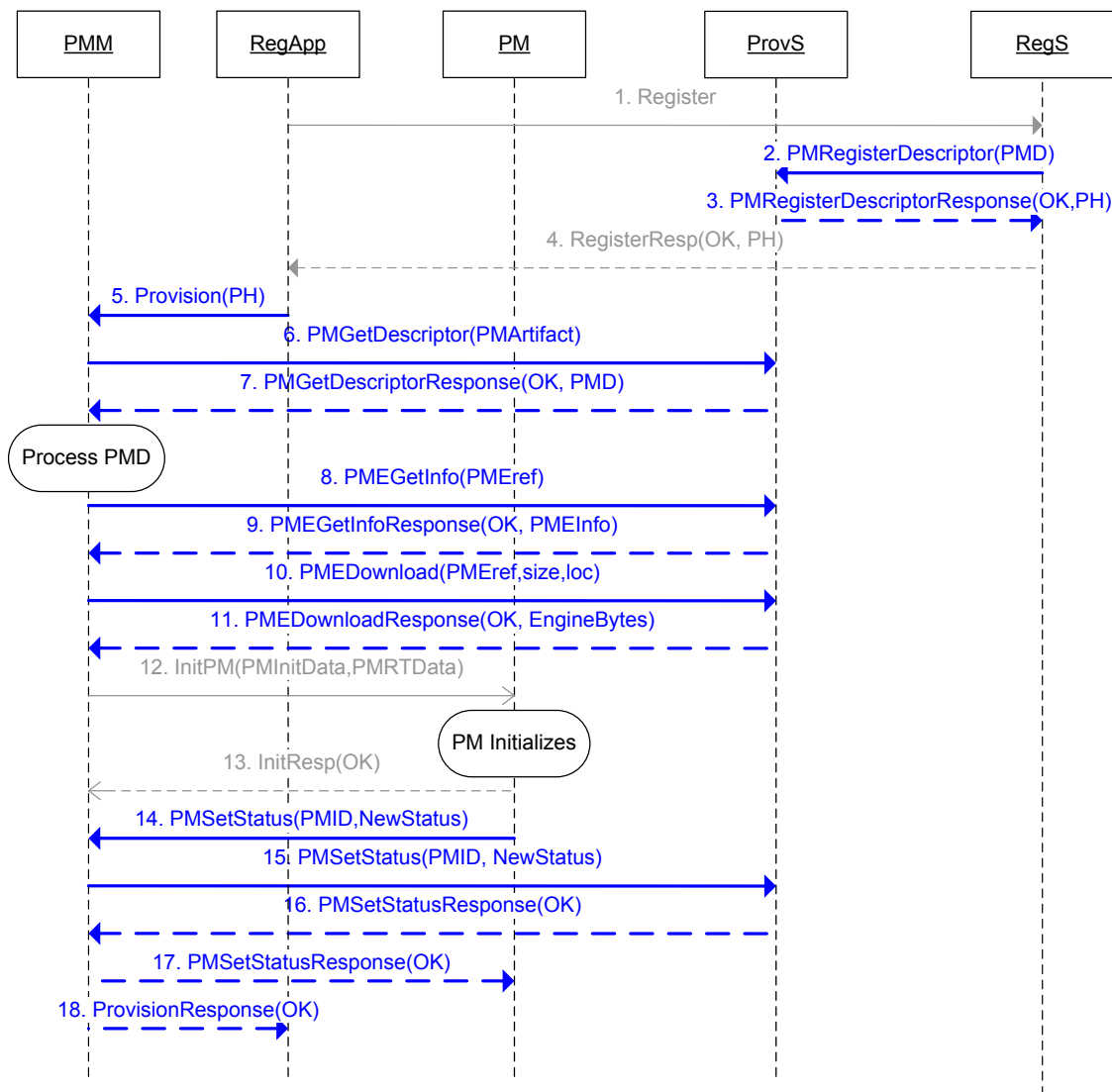
861 The provisioning system defined by Liberty allows for many different workflows to support many different implemen-  
 862 tations. We will walk through two different scenarios to show how the different components can work together to  
 863 support the provisioning process.

#### 864 5.3.1. Initial Provisioning

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

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

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



871

872

Figure 7. Provisioning Workflow

873 Things to note about the diagram:

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

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

#### 879 **5.3.1.1. Step 1: User registration.**

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

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

#### 884 **5.3.1.2. Step 2: PMD Registration at ProvS.**

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

```
887 <prov:PMRegisterDescriptor>
888   <prov:PMRegisterDescriptorItem itemID="1">
889     <prov:PMDescriptor xs:id="2323923900239">
890       <prov:PMID issuer="http://provs-r-us.com">uuid:239032-230328-92379-2397923</prov:PMID>
891       <prov:PMEngineRef>https://pmsRus.org/VeryTrustedModule/3.7</prov:PMEngineRef>
892       <prov:PMInitData>
893         <MyData>
894           .... initialization data here ...
895         </MyData>
896       </prov:PMInitData>
897       <ds:Signature>
898         ... signature data goes here ...
899       </ds:Signature>
900     </prov:PMDescriptor>
901   </prov:PMRegisterDescriptorItem>
902 </prov:PMRegisterDescriptor>
903
```

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

#### 905 **5.3.1.3. Step 3: ProvS responds to RegS with PH.**

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

```

908 <prov:PMRegisterDescriptorResponse>
909   <lu:Status code="OK" />
910   <prov:PMRegisterDescriptorResponseItem ref="1">
911     <prov:ProvisioningHandle xs:id="2302384823023">
912       <prov:PMDArtifact>23asdfhoi323hpos df923h9sdfhweorh2398asdfjweoiha</prov:PMDArtifact>
913       <prov:ProvisioningServiceEPR>
914         <wsa:Address>http://provision.idpsRus.com</wsa:Address>
915         <wsa:Metadata>
916           <ds:Abstract>Provisioning Service</ds:Abstract>
917           <ds:ProviderID>http://provisioning-provider.idpsRus.com/</ds:ProviderID>
918           <ds:ServiceType>urn:liberty:prov:2007-09</ds:ServiceType>
919           <ds:Framework version="2.0" />
920           <ds:SecurityContext>
921             <ds:SecurityMechID>
922               urn:liberty:security:2005-02:TLS:SAMLV2
923             </ds:SecurityMechID>
924             <sec:Token ref="urn:liberty:disco:tokenref:ObtainFromIDP" />
925           </ds:SecurityContext>
926         </wsa:Metadata>
927       </prov:ProvisioningServiceEPR>
928       <ds:Signature>
929         ... signature info here ..
930       </ds:Signature>
931     </prov:ProvisioningHandle>
932   </prov:PMRegisterDescriptorResponseItem>
933 </prov:PMRegisterDescriptorResponse>
934

```

935 **Example 25.** <prov:PMRegisterDescriptorResponse> Message

#### 936 5.3.1.4. Step 4: RegS sends PH to RegApp.

937 The RegS sends the PH to the RegApp.

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

#### 939 5.3.1.5. Step 5: RegApp initiates Provisioning at PMM.

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

```

942 <pmm:Provision wait="true" >
943   <prov:ProvisioningHandle xs:id="2302384823023">
944     <prov:PMDArtifact>23asdfhoi323hpos df923h9sdfhweorh2398asdfjweoiha</prov:PMDArtifact>
945     <prov:ProvisioningServiceEPR>
946       <wsa:Address>http://provision.idpsRus.com</wsa:Address>
947       <wsa:Metadata>
948         <ds:Abstract>Provisioning Service</ds:Abstract>
949         <ds:ProviderID>http://provisioning-provider.idpsRus.com/</ds:ProviderID>
950         <ds:ServiceType>urn:liberty:prov:2007-09</ds:ServiceType>
951         <ds:Framework version="2.0" />
952         <ds:SecurityContext>
953           <ds:SecurityMechID>
954             urn:liberty:security:2005-02:TLS:SAMLV2
955           </ds:SecurityMechID>
956           <sec:Token ref="urn:liberty:disco:tokenref:ObtainFromIDP" />
957         </ds:SecurityContext>
958       </wsa:Metadata>
959     </prov:ProvisioningServiceEPR>
960     <ds:Signature>
961       ... signature info here ..
962     </ds:Signature>
963   </prov:ProvisioningHandle>
964 </pmm:Provision>
965

```

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

967 **5.3.1.6. Step 6: PMM Asks ProvS for the PMD associated with PH.**

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

```
970 <prov:PMGetDescriptor>  
971 <prov:PMDArtifact>23asdfhoi323hpo sdf923h9sdfhweorh2398asdfjweoi ha</prov:PMDArtifact>  
972 <prov:CallbackEPR>  
973 <wsa:Address>http://www.w3.org/2005/08/ addressing/anonymous</wsa:Address>  
974 </prov:CallbackEPR>  
975 </prov:PMGetDescriptor>  
976
```

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

978 **5.3.1.7. Step 7: ProvS responds to the PMM with the PMD.**

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

```
981 <prov:PMGetDescriptorResponse >  
982 <lu:Status code="OK" />  
983 <prov:PMDDescriptor xs:id="2323923900239">  
984 <prov:PMID issuer="http://provs-r-us.com">uuid:239032-230328-92379-2 397923</prov:PMID>  
985 <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/3.7</prov:PMEngineRef>  
986 <prov:PMInitData>  
987 <MyData>  
988 .... initialization data here ...  
989 </MyData>  
990 </prov:PMInitData>  
991 <ds:Signature>  
992 ... signature data goes here ...  
993 </ds:Signature>  
994 </prov:PMDDescriptor>  
995 </prov:PMGetDescriptorResponse>  
996
```

997 **Example 28.** `<prov:PMGetDescriptorResponse>` Message

998 **5.3.1.8. Step 8: PMM retrieves PME information from the ProvS.**

999 The PMM parses the PMD and noting that it does not have an instance of that PME invokes the `<prov:PMEGetInfo>`  
1000 interface to obtain the PME information (size, hash, etc).

1001 This request would **not** be necessary if the PMM already had an instance of the PME available locally (perhaps because  
1002 another PM was instantiated which used the same PME, or because that particular PME was built into the platform).

1003 This request would look something like:

```
1004 <prov:PMEGetInfo>  
1005 <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>  
1006 </prov:PMEGetInfo>  
1007
```

1008 **Example 29.** `<prov:PMEGetInfo>` Request Message

1009 **5.3.1.9. Step 9: ProvS responds to the PMM with the PME Info.**

1010 The ProvS returns the PME Information to the PMM. This response would look something like:

```
1011 <prov:PMEGetInfoResponse>
1012   <lu:Status code="OK" />
1013   <prov:PMEInfo>
1014     <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
1015     <prov:PMECreatorID>http://reg.providers.com</prov:PMECreatorID>
1016     <prov:PMEWhenCreated>2007-01-18T17:32:14Z</prov:PMEWhenCreated>
1017     <prov:PMEEnabled>true</prov:PMEEnabled>
1018     <prov:PMEWhenEnabled>2007-01-18T18:15:22Z</prov:PMEWhenEnabled>
1019     <prov:PMESize>185676</prov:PMESize>
1020     <prov:PMEHash method="...SHA-1">...SHA1 hash data ...</prov:PMEHash>
1021   </prov:PMEInfo>
1022 </prov:PMEGetInfoResponse>
1023
```

1024 **Example 30. <prov:PMEGetInfoResponse> Message**

1025 **5.3.1.10. Step 10: PMM retrieves the PME from the ProvS.**

1026 The PMM initiates the download of the PME from the ProvS using the <prov:PMEDownload> interface.

1027 This PMM specifies how much of the PME it wants to get in each <prov:PMEDownload> invocation and will usually  
1028 invoke this several times to download the entire executable.

1029 This request would look something like:

```
1030 <prov:PMEDownload count="102400">
1031   <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
1032 </prov:PMEDownload>
1033
```

1034 **Example 31. <prov:PMEDownload> Request Message**

1035 **5.3.1.11. Step 11: ProvS responds to the PMM with the PME.**

1036 The ProvS returns a portion of the PME. This sequence can be repeated multiple times as the PMM pulls sections of  
1037 the PME rather than downloading the entire PME in a single call. This response would look something like:

```
1038 <prov:PMEDownloadResponse remaining="83276" nextOffset="102400">
1039   <lu:Status code="OK"/>
1040   <prov:EngineData>
1041     ... base64 encoded data (100K worth) ...
1042   </prov:EngineData>
1043 </prov:PMEDownloadResponse>
1044
```

1045 **Example 32. <prov:PMEDownloadResponse> Message**

1046 **5.3.1.12. Step 12: PMM Initialized the PM.**

1047 The PMM does what magic it must do to instantiate the PM within the trusted container. This may involve sending  
1048 a message to a PME Init interface, or it may involve just starting up the PME executable (passing in the PMInitData  
1049 and/or PMRTData as parameters).



1050 This message and the method chosen by the PM is out-of-scope for Liberty and hence there is no Liberty ID-WSF  
1051 defined interface to accomplish this task.

1052 **5.3.1.13. Step 13: PM Init routine returns OK.**

1053 The PM acknowledges receipt of the initialization parameters. This is another out-of-scope, non-Liberty message and  
1054 may not even take place in some environments.

1055 **5.3.1.14. Step 14: PM sets its current status to Active.**

1056 The PM, upon completion of its initialization and verification of the passed in PMInitData and PMRTData (if any),  
1057 uses the `<pmm:PMSetStatus>` interface to set its current status to `urn:liberty:prov:2007-09:status:Active`.

1058 Some implementations might use internal proprietary messages to accomplish this task (that would tightly bind the  
1059 PM to a particular instance of a PMM) and others will use the Liberty defined message. That is an implementation  
1060 decision.

1061 In the case of using the Liberty defined message, the PM knows where to send this information and the PMID  
1062 associated with the PM through some out-of-scope means (such as it being passed as an input parameter during  
1063 the initialization stage).

1064 This request would look something like:

```
1065 <pmm:PMSetStatus>  
1066   <prov:PMStatus>  
1067     <prov:PMID issuer="http://provs-r-us.com">uuid:778349-283920-88379-5448739</prov:PMID>  
1068     <prov:State>urn:liberty:prov:2007-09:status:Activated</prov:State>  
1069   </prov:PMStatus>  
1070 </pmm:PMSetStatus>  
1071
```

1072 **Example 33. `<pmm:PMSetStatus>` Request Message**

1073 **5.3.1.15. Step 15: PMM updates the PMD status at the ProvsS.**

1074 The PMM uses the `<prov:PMSetStatus>` interface to update the status of the PMD at the ProvsS.

1075 This request would look something like:

```
1076 <prov:PMSetStatus>  
1077   <prov:PMStatus>  
1078     <prov:PMID issuer="http://provs-r-us.com">uuid:239032-230328-92379-2397923</prov:PMID>  
1079     <prov:State>urn:liberty:prov:2007-09:status:Active</prov:State>  
1080   </prov:PMStatus>  
1081 </prov:PMSetStatus>  
1082
```

1083 **Example 34. `<prov:PMSetStatus>` Request Message**

1084 **5.3.1.16. Step 16: ProvsS responds to the PMM with an OK response.**

1085 The ProvsS responds with an OK.

1086 This message would look something like:

```
1087 <prov:PMSetStatusResponse>
1088   <lu:Status code="OK" />
1089 </prov:PMSetStatusResponse>
1090
```

1091 **Example 35.** `<prov:PMDSetStatusResponse>` Message

### 1092 **5.3.1.17. Step 17: PMM responds to status change with OK.**

1093 The PMM responds to the status update made in step 14 ([Section 5.3.1.14](#)) with an OK.

1094 In this particular sequence the OK response is made after the PMM has updated the ProvS status ([Section 5.3.1.15](#));  
1095 however, there is no normative requirement for this particular ordering. The PMM could have sent this OK response  
1096 immediately after the `<pmm:PMSetStatus>` request and then went on to update the status at the ProvS.

1097 In any case, this response would look something like:

```
1098 <pmm:PMSetStatusResponse>
1099   <lu:Status code="OK" />
1100 </pmm:PMSetStatusResponse>
1101
```

1102 **Example 36.** `<pmm:PMSetStatusResponse>` Message

### 1103 **5.3.1.18. Step 18: PMM responds to RegApp with an OK.**

1104 The PMM responds to the provision request in step 5 ([Section 5.3.1.5](#)) with an OK.

1105 The timing of this response is somewhat arbitrary. However, it should not take place until the PMM is satisfied that  
1106 the PM is instantiated and available (at which point the RegApp could make use of it). So, this message could have  
1107 been sent anytime after receipt of the status update in step 14 ([Section 5.3.1.14](#)).

1108 This response would look something like:

```
1109 <pmm:ProvisionResponse>
1110   <lu:Status code="OK" />
1111 </pmm:ProvisionResponse>
1112
```

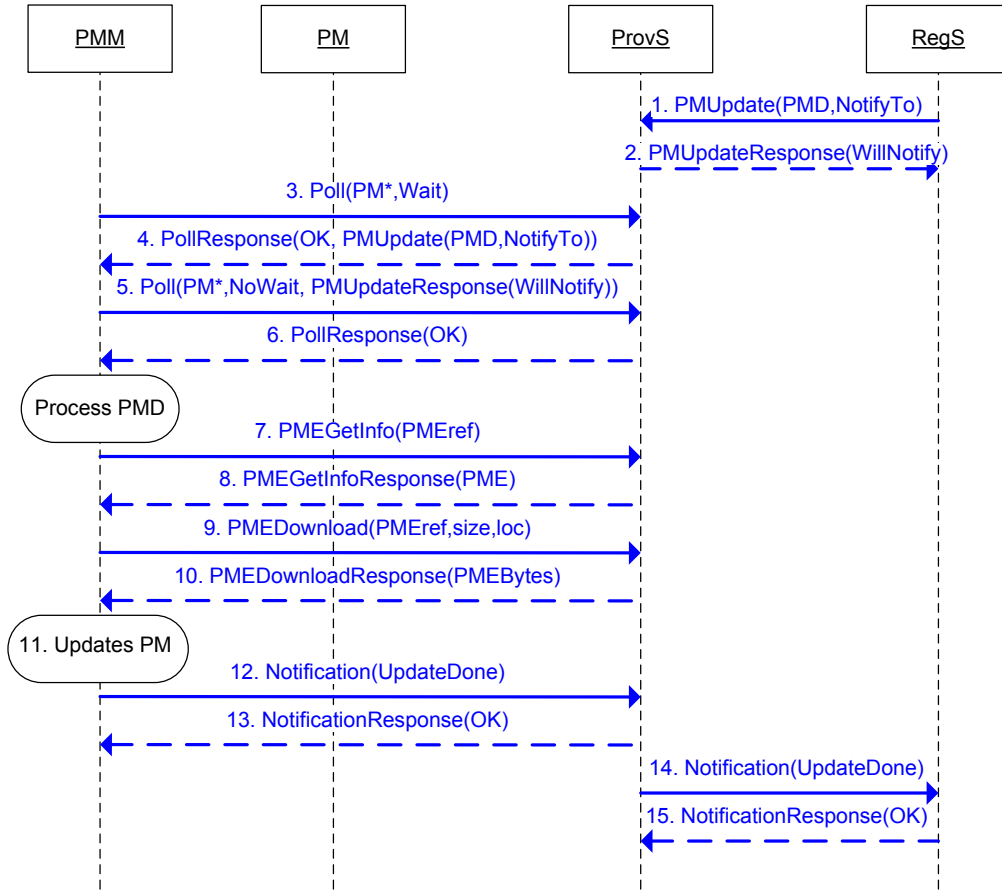
1113 **Example 37.** `<pmm:ProvisionResponse>` Message

## 1114 **5.3.2. Update Provisioning**

1115 This use case walks through a potential sequence of steps to update an existing PM (previously provisioned) on a client  
1116 platform. In this case, the PMM is not directly addressable by the ProvS and as such, the ProvS must wait for the  
1117 PMM to poll it for any update request.

1118 The RegS is the controlling entity in this use case, building and issuing the update request through the ProvS. The  
1119 PMID MUST be the same PMID that was used when the PM was originally provisioned (so that the ProvS will know  
1120 which PMM to send the request to and so that the PMM will know which PM is being updated).

1121 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:

```
1127 <prov:PMUpdate>
1128   <prov:PMUpdateItem itemID="1" type="urn:liberty:prov:2007-09:ut:engine">
1129     <prov:PMDescriptor xs:id="2323923900239" >
1130       <prov:PMID issuer="http://provs-r-us.com">uuid:778349-283920-88379-5448739</prov:PMID>
1131       <prov:PMEngineRef>https://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
1132       <ds:Signature>
1133         ... signature data goes here ...
1134       </ds:Signature>
1135     </prov:PMDescriptor>
1136   </prov:PMUpdateItem>
1137   <dp:NotifyTo>
1138     <wsa:Address>https://provider.com/notifications</wsa:Address>
1139     <wsa:Metadata>
1140       <ds:ProviderID>http://provider.com/</ds:ProviderID>
1141       <ds:ServiceType>urn:liberty:dp:2007-09:notification</ds:ServiceType>
1142       <ds:Framework version="2.0" />
1143       <ds:SecurityContext>
1144         <ds:SecurityMechID>urn:liberty:security:2005-02:TLS:null</ds:SecurityMechID>
1145       </ds:SecurityContext>
1146     </wsa:Metadata>
1147   </dp:NotifyTo>
1148 </prov:PMUpdate>
1149
```

1150 **Example 38. <prov:PMUpdate> Request Message**

### 1151 **5.3.2.2. Step 2: ProvS responds to RegS with WillNotify.**

1152 The ProvS acknowledges receipt of the update request. The ProvS must verify that it already has the correct PME for  
1153 the updated PMD prior to acknowledging the receipt (or, if ProvS does not have the correct PME, it should return an  
1154 error).

1155 The RegS can use the <prov:PMEUpload> call to send the PME to the ProvS, if necessary.

1156 The ProvS acknowledgement will look something like:

```
1157 <prov:PMUpdateResponse>
1158   <lu:Status code="WillNotify"/>
1159 </prov:PMUpdateResponse>
1160
```

1161 **Example 39. <prov:PMDUpdateResponse> Message**

### 1162 **5.3.2.3. Step 3: The PMM polls the ProvS for any requests.**

1163 The PMM in this case, is unable to expose a network accessible interface for any incoming requests and so must  
1164 periodically poll the ProvS for any outstanding requests using the <prov:Poll> request. The frequency of these  
1165 requests is under control of the ProvS and the PMM.

1166 In this particular case, the PMM just happened to poll the ProvS shortly after the ProvS received the update request.  
1167 In other cases the PMM may already have a waiting poll request (i.e., this message may have sent to the ProvS before  
1168 the update request from the RegS) or the poll request could take place much later, depending upon the polling cycle.

1169 This request would look something like:

```
1170 <prov:Poll wait="300">
1171   <wsa:Action>urn:liberty:pmm:2007-09:PMUpdate</wsa:Action>
1172   <wsa:Action>urn:liberty:pmm:2007-09:PMDelete</wsa:Action>
1173   <wsa:Action>urn:liberty:pmm:2007-09:PMGetStatus</wsa:Action>
1174 </prov:Poll>
1175
```

1176 **Example 40. <prov:Poll> Request Message**

#### 1177 **5.3.2.4. Step 4: ProvS responds to the PMM with PMUpdate request.**

1178 The ProvS responds to the <prov:Poll> request with an OK status and includes the <pmm:PMUpdate> request for  
1179 the PMM asking to be notified of the completion of the update at the same endpoint where the polling requests are  
1180 delivered.

1181 This response would look something like:

```
1182 <prov:PollResponse>
1183   <lu:Status code="OK" />
1184   <dp:Request itemID="1">
1185     <pmm:PMUpdate>
1186       <pmm:PMUpdateItem itemID="1" type="urn:liberty:prov:2007-09:ut:engine">
1187         <prov:PMDescriptor xs:id="2323923900239" >
1188           <prov:PMID issuer="http://provs-r-us.com">uuid:778349-2 83920-88379-5448739</prov:PMID>
1189           <prov:PMEngineRef>https://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
1190           <ds:Signature>
1191             ... signature data goes here ...
1192           </ds:Signature>
1193         </prov:PMDescriptor>
1194       </pmm:PMUpdateItem>
1195       <dp:NotifyTo>
1196         <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
1197       </dp:NotifyTo>
1198     </pmm:PMUpdate>
1199   </dp:Request>
1200 </prov:PollResponse>
1201
1202
```

1203 **Example 41. <prov:PollResponse> Message**

#### 1204 **5.3.2.5. Step 5: The PMM polls the ProvS again to send PMUpdateResponse.**

1205 The PMM immediately polls the ProvS again, this time including the response to the <pmm:PMUpdate> request that  
1206 was included in the <prov:PollResponse> in step 4 ([Section 5.3.2.4](#)) above.

1207 This poll, while accepting new requests, also tells the ProvS to not wait for new work, but return immediately, even if  
1208 there are no new requests.

1209 This request would look something like:

```
1210 <prov:Poll wait="300">
1211   <wsa:Action>urn:liberty:pmm:2007-09:PMUpdate</wsa:Action>
1212   <wsa:Action>urn:liberty:pmm:2007-09:PMDelete</wsa:Action>
1213   <wsa:Action>urn:liberty:pmm:2007-09:PMGetStatus</wsa:Action>
1214   <dp:Response ref="1">
1215     <prov:UpdatePMResponse>
1216       <lu:Status code="WillNotify" />
1217     </prov:UpdatePMResponse>
1218   </dp:Response>
1219 </prov:Poll>
1220
```

1221 **Example 42. <prov:Poll> Request Message**

### 1222 5.3.2.6. Step 6: ProvS responds to the PMM with OK.

1223 The ProvS responds to the <prov:Poll> request with an OK status and since there is no other outstanding request,  
1224 nothing else. The ProvS also tells the PMM to poll again in 10 minutes. This response would look something like:

```
1225 <prov:PollResponse nextPoll="600">
1226   <lu:Status code="OK" />
1227 </prov:PollResponse>
1228
```

1229 **Example 43. <prov:PollResponse> Message**

### 1230 5.3.2.7. Step 7: PMM retrieves PME information from the ProvS.

1231 The PMM parses the PMD and noting that it does not have an instance of that PME invokes the <prov:PMEGetInfo>  
1232 interface to obtain the PME information (size, hash, etc).

1233 This request would **not** be necessary if the PMM already had an instance of the PME available locally (perhaps because  
1234 another PM was instantiated which used the same PME, or because that particular PME was built into the platform).

1235 This request would look something like:

```
1236 <prov:PMEGetInfo>
1237   <prov:PMEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEngineRef>
1238 </prov:PMEGetInfo>
1239
```

1240 **Example 44. <prov:PMEGetInfo> Request Message**

### 1241 5.3.2.8. Step 8: ProvS responds to the PMM with the PME Info.

1242 The ProvS returns the PME Information to the PMM. This response would look something like:

```
1243 <prov:PMEGetInfoResponse>
1244   <lu:Status code="OK" />
1245   <prov:PMEInfo>
1246     <prov:PMEEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEEngineRef>
1247     <prov:PMECreatorID>http://reg.providers.com</prov:PMECreatorID>
1248     <prov:PMEWhenCreated>2007-01-18T17:32:14Z</prov:PMEWhenCreated>
1249     <prov:PMEEnabled>true</prov:PMEEnabled>
1250     <prov:PMEWhenEnabled>2007-01-18T18:15:22Z</prov:PMEWhenEnabled>
1251     <prov:PMESize>185676</prov:PMESize>
1252     <prov:PMEHash method="...SHA-1">...SHA1 hash data ...</prov:PMEHash>
1253   </prov:PMEInfo>
1254 </prov:PMEGetInfoResponse>
1255
```

1256 **Example 45. <prov:PMEGetInfoResponse> Message**

### 1257 **5.3.2.9. Step 9: PMM retrieves the updated PME from the ProvS.**

1258 The PMM initiates the download of the PME from the ProvS using the <prov:PMEDownload> interface.

1259 This PMM specifies how much of the PME it wants to get in each <prov:PMEDownload> invocation and will usually  
1260 invoke this several times to download the entire executable.

1261 This request would look something like:

```
1262 <prov:PMEDownload count="102400">
1263   <prov:PMEEngineRef>http://pmsRus.org/VeryTrustedModule/4.0</prov:PMEEngineRef>
1264 </prov:PMEDownload>
1265
```

1266 **Example 46. <prov:PMEDownload> Request Message**

### 1267 **5.3.2.10. Step 10: ProvS responds to the PMM with the PME.**

1268 The ProvS returns a portion of the PME. This sequence can be repeated multiple times as the PMM pulls sections of  
1269 the PME rather than downloading the entire PME in a single call. This response would look something like:

```
1270 <prov:PMEDownloadResponse remaining="83276" nextOffset="102400">
1271   <lu:Status code="OK"/>
1272   <prov:EngineData>
1273     ... base64 encoded data (100K worth) ...
1274   </prov:EngineData>
1275 </prov:PMEDownloadResponse>
1276
```

1277 **Example 47. <prov:PMEDownloadResponse> Message**

### 1278 **5.3.2.11. Step 11: PMM updates the PM.**

1279 The PMM updates the PM. The actual mechanisms used to perform the update are out-of-scope for Liberty and  
1280 therefore no messages or explicit steps are shown.

### 1281 **5.3.2.12. Step 12: PMM Notifies ProvS about update completion.**

1282 The PMM, upon completion of the update of the PM, uses the <dp:Notification> interface to notify the ProvS of  
1283 the completion status of the update request (in this case a success).

1284 This request would look something like:

```
1285 <dp:Notification ref="...messageID-of-request...">
1286   <pmm:PMUpdateResponse>
1287     <lu>Status code="OK" />
1288   </pmm:PMUpdateResponse>
1289 </dp:Notification>
1290
1291
```

1292 **Example 48. <dp:Notification> Request Message**

### 1293 **5.3.2.13. Step 13: ProvS responds to notification with OK.**

1294 The ProvS responds to the notification with an OK.

1295 In this particular sequence the OK response is made before the ProvS has sent the completion status notification to the  
1296 original requester. However, there is no normative requirement for this particular ordering. The ProvS could have  
1297 sent this OK response after it had sent the subsequent notification to the original requester.

1298 In any case, this response would look something like:

```
1299 <dp:NotificationResponse>
1300   <lu>Status code="OK" />
1301 </dp:NotificationResponse>
1302
```

1303 **Example 49. <dp:NotificationResponse> Message**

### 1304 **5.3.2.14. Step 14: ProvS Notifies RegS about update completion.**

1305 The ProvS, upon receipt of the update notification from the PMM, sends its own notification to the RegS with the  
1306 completion status of the request (in this case a success).

1307 This request would look something like:

```
1308 <dp:Notification ref="...messageID-of-request...">
1309   <prov:PMUpdateResponse>
1310     <lu>Status code="OK"/>
1311   </prov:PMUpdateResponse>
1312 </dp:Notification>
1313
```

1314 **Example 50. <dp:Notification> Request Message**

### 1315 **5.3.2.15. Step 15: RegS responds to notification with OK.**

1316 The RegS responds to the notification with an OK.

1317 This response would look something like:

```
1318 <dp:NotificationResponse>
1319   <lu>Status code="OK" />
1320 </dp:NotificationResponse>
1321
```

1322 **Example 51. <dp:NotificationResponse> Message**



## 1323 References

### 1324 Normative

- 1325 [LibertyIdP] Cahill, Conor P., eds. "Liberty ID-WSF IDP Service Specification," Version 1.0, Liberty Alliance Project  
1326 (14 December 2007). <http://www.projectliberty.org/specs>
- 1327 [LibertySHPS] Cahill, Conor, eds. "Liberty ID-WSF Service Hosting and Proxying Service Specification," Version  
1328 1.0, Liberty Alliance Project (14 December 2007). <http://www.projectliberty.org/specs>
- 1329 [LibertyPROV] Cahill, Conor P., eds. "Liberty ID-WSF Provisioning Service Specification," Version 1.0, Liberty  
1330 Alliance Project (14 December 2007). <http://www.projectliberty.org/specs>
- 1331 [LibertyPMM] Cahill, Conor P., eds. "Liberty ID-WSF Provisioned Module Manager Service Specification," Version  
1332 1.0, Liberty Alliance Project (14 December 2007). <http://www.projectliberty.org/specs>
- 1333 [LibertyDisco] Cahill, Conor, Hodges, Jeff, eds. "Liberty ID-WSF Discovery Service Specification," Version 2.0-  
1334 errata-v1.0, Liberty Alliance Project (29 November, 2006). <http://www.projectliberty.org/specs>
- 1335 [LibertyIDPP] Kellomäki, Sampo, Lockhart, Rob, eds. "Liberty ID-SIS Personal Profile Service Specification,"  
1336 Version 1.1, Liberty Alliance Project (29 September, 2005). <http://www.projectliberty.org/specs>
- 1337 [LibertyInteract] Aarts, Robert, Madsen, Paul, eds. "Liberty ID-WSF Interaction Service Specification," Version 2.0-  
1338 errata-v1.0, Liberty Alliance Project (21 April, 2007). <http://www.projectliberty.org/specs>
- 1339 [LibertyProtSchema] Cantor, Scott, Kemp, John, eds. "Liberty ID-FF Protocols and Schema Specification," Version  
1340 1.2-errata-v3.0, Liberty Alliance Project (14 December 2004). <http://www.projectliberty.org/specs>
- 1341 [LibertyReg] Kemp, John, eds. "Liberty Enumeration Registry Governance," Version 1.1, Liberty Alliance Project (14  
1342 December, 2004). <http://www.projectliberty.org/specs>
- 1343 [LibertySOAPAuthn] Hodges, Jeff, Aarts, Robert, Madsen, Paul, Cantor, Scott, eds. "Liberty ID-WSF Authentication,  
1344 Single Sign-On, and Identity Mapping Services Specification," v2.0-errata-1.0-01, Liberty Alliance Project  
1345 (28 November, 2006). <http://www.projectliberty.org/specs>
- 1346 [LibertySOAPBinding] Hodges, Jeff, Kemp, John, Aarts, Robert, Whitehead, Greg, Madsen, Paul, eds. "Liberty  
1347 ID-WSF SOAP Binding Specification," Version 2.0-errata-v1.0, Liberty Alliance Project (21 April, 2007).  
1348 <http://www.projectliberty.org/specs>
- 1349 [LibertyPAOS] Aarts, Robert, Kemp, John, eds. "Liberty Reverse HTTP Binding for SOAP Specification," Version  
1350 2.0, Liberty Alliance Project (30 July, 2006). <http://www.projectliberty.org/specs>
- 1351 [RFC2119] S. Bradner "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119, The Internet  
1352 Engineering Task Force (March 1997). <http://www.ietf.org/rfc/rfc2119.txt>
- 1353 [RFC2251] "Lightweight Directory Access Protocol (v3)," M. Wahl T. Howes S. Kille (December 1997). RFC 2251,  
1354 Internet Engineering Task Force <http://www.ietf.org/rfc/rfc2251.txt>
- 1355 [RFC2252] "Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions," M. Wahl A.  
1356 Coulbeck T. Howes S. Kille (December 1997). RFC 2252, Internet Engineering Task Force  
1357 <http://www.ietf.org/rfc/rfc2252.txt>
- 1358 [RFC2891] Howes, T., Wahl, M., eds. (August 2000). "LDAP Control Extension for Server Side Sorting of Search  
1359 Results," RFC 2891, Internet Engineering Task Force <http://www.ietf.org/rfc/rfc2891.txt>

- 1360 [SAMLCore11] Maler, Eve, Mishra, Prateek, Philpott, Rob, eds. (2 September 2003). "Assertions and Pro-  
1361        protocol for the OASIS Security Assertion Markup Language (SAML) V1.1," SAML v1.1, OASIS  
1362        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](http://www.oasis-<br/>1363        open.org/committees/download.php/3406/oasis-sstc-saml-core-1.1.pdf)
- 1364 [SAMLCore2] Cantor, Scott, Kemp, John, Philpott, Rob, Maler, Eve, eds. (15 March 2005). "Assertions  
1365        and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0," SAML V2.0, OA-  
1366        SIS Standard, Organization for the Advancement of Structured Information Standards [http://docs.oasis-  
open.org/security/saml/v2.0/saml-core-2.0-os.pdf](http://docs.oasis-<br/>1367        open.org/security/saml/v2.0/saml-core-2.0-os.pdf)
- 1368 [SAMLProf2] Hughes, John, Cantor, Scott, Hodges, Jeff, Hirsch, Frederick, Mishra, Prateek, Philpott, Rob, Maler,  
1369        Eve, eds. (15 March, 2005). "Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0,"  
1370        SAML V2.0, OASIS Standard, Organization for the Advancement of Structured Information Standards  
1371        <http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf>
- 1372 [Schema1-2] Thompson, Henry S., Beech, David, Maloney, Murray, Mendelsohn, Noah, eds. (28 October  
1373        2004). "XML Schema Part 1: Structures Second Edition," Recommendation, World Wide Web Consortium  
1374        <http://www.w3.org/TR/xmlschema-1/>
- 1375 [SOAPv1.1] "Simple Object Access Protocol (SOAP) 1.1," Box, Don, Ehnebuske, David , Kakivaya, Gopal, Layman,  
1376        Andrew, Mendelsohn, Noah, Nielsen, Henrik Frystyk, Winer, Dave, eds. World Wide Web Consortium W3C  
1377        Note (08 May 2000). <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>
- 1378 [XML] Bray, Tim, Paoli, Jean, Sperberg-McQueen, C. M., Maler, Eve, Yergeau, Francois, eds. (04 February 2004).  
1379        "Extensible Markup Language (XML) 1.0 (Third Edition)," Recommendation, World Wide Web Consortium  
1380        <http://www.w3.org/TR/2004/REC-xml-20040204>
- 1381 [XMLDsig] Eastlake, Donald, Reagle, Joseph, Solo, David, eds. (12 Feb 2002). "XML-Signature Syntax and  
1382        Processing," Recommendation, World Wide Web Consortium <http://www.w3.org/TR/xmlsig-core>
- 1383 [XPATH] Clark , J., DeRose , S., eds. (16 November 1999). "XML Path Language (XPath) Version 1.0 ,"  
1384        Recommendation, W3C <http://www.w3.org/TR/xpath> [August 2003].

## 1385 Informative

- 1386 [LibertyIDWSFGuide] Weitzel, David, eds. "Liberty ID-WSF Implementation Guide," Version 2.0-02, Liberty  
1387        Alliance Project (13 January, 2005). <http://www.projectliberty.org/specs>
- 1388 [LibertyIDPPGuide] Kellomäki, Sampo, Lockhart, Rob, eds. "Liberty ID-SIS Personal Profile Service Implementation  
1389        Guidelines," Version 1.1, Liberty Alliance Project (29 September, 2005). <http://www.projectliberty.org/specs>
- 1390 [LibertyIDWSFOverview] Tourzan, Jonathan, Koga, Yuzo, eds. "Liberty ID-WSF Web Services Framework  
1391        Overview," Version 2.0, Liberty Alliance Project (30 July, 2006). <http://www.projectliberty.org/specs>