Liberty ID-WSF Subscriptions and Notifications

Version: 1.0-05

Editors:
Sampo Kellomäki, Symlabs, Inc.

Contributors:
Robert Aarts, Hewlett-Packard
Rajeev Angal, Sun Microsystems, Inc.
Conor Cahill, America Online, Inc.
Carolina Canales-Valenzuela, Ericsson
Darryl Champagne, IEEE-ISTO
Andy Feng, America Online, Inc.
Gael Gourmelen, France Télécom
Jeff Hodges, NeuStar, Inc.
Jukka Kainulainen, Nokia Corp.
Lena Kannappan, France Télécom
John Kemp, Nokia Corporation
Rob Lockhart, IEEE-ISTO
Paul Madsen, NTT
Aravindan Ranganathan, Sun Microsystems, Inc.
Matti Saarenpää, Nokia Corporation
Jonathan Sergent, Sun Microsystems, Inc.
Lakshmanan Suryanarayanan, America Online, Inc
Greg Whitehead, Hewlett-Packard

Abstract:

This specification provides protocols for subscription and notification. A subscription is a mechanism by which a WSC can register to receive notifications from a WSP when some data changes or some event happens. The subscriptions and notifications are applicable to any ID-WSF based service, but specific guidance is provided on how to apply them on a DST based service.

Filename: draft-liberty-idwsf-subs-v1.0-05.pdf
Notice

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

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

Copyright © 2006 Adobe Systems; America Online, Inc.; American Express Company; Amsoft Systems Pvt Ltd.;
Avatier Corporation; Axalto; Bank of America Corporation; BIPAC; BMC Software, Inc.; Computer Associates
International, Inc.; DataPower Technology, Inc.; Diversinet Corp.; Enosis Group LLC; Entrust, Inc.; Epok, Inc.;
Ericsson; Fidelity Investments; Forum Systems, Inc.; France Télécom; French Government Agence pour le
développement de l’administration électronique (ADAE); Gamfederation; Gemplus; General Motors; Giesecke &
Devrient GmbH; GSA Office of Governmentwide Policy; Hewlett-Packard Company; IBM Corporation; Intel
Corporation; Intuit Inc.; Kantega; Kayak Interactive; MasterCard International; Mobile Telephone Networks (Pty)
Ltd; NEC Corporation; Netegrity, Inc.; NeuStar, Inc.; Nippon Telegraph and Telephone Corporation; Nokia
Corporation; Novell, Inc.; NTT DoCoMo, Inc.; OpenNetwork; Oracle Corporation; Ping Identity Corporation;
Reactivity Inc.; Royal Mail Group plc; RSA Security Inc.; SAP AG; Senforce; Sharp Laboratories of America;
Sigaba; SmartTrust; Sony Corporation; Sun Microsystems, Inc.; Supremacy Financial Corporation; Symlabs, Inc.;
Telecom Italia S.p.A.; Telefónica Móviles, S.A.; Trusted Network Technologies; Trustgenix; UTI; VeriSign, Inc.;
Vodafone Group Plc.; Wave Systems Corp. All rights reserved.
Contents

1. Introduction .................................................. 4
   1.1. Notation ................................................ 4
   1.2. Liberty Considerations ................................ 4
   1.3. Namespaces .............................................. 4
   1.4. Applying Subscriptions to DST Based Services .......... 4
2. General Rules Regarding Subscriptions and Notifications ............. 6
   2.1. Second level <Status> codes for Subscriptions .......... 6
   2.2. Discovery Option Keywords for Indicating Lack of Subscription Support .......... 6
   2.3. CRUD Manipulation using Object Type "Subscription" .......... 6
   2.4. No itemIDRef for subscription related <ItemData> .......... 6
3. Piggy Backing Subscriptions to DST Operations ............................. 7
   3.1. <Query> with <Subscription> ................................ 7
   3.2. <Create> with <Subscription> ................................ 7
   3.3. <Modify> with <Subscription> ................................ 7
4. Subscriptions ................................................................ 8
   4.1. `<Subscription>` element ..................................... 8
   4.2. Selecting data a subscription applies to ....................... 9
   4.3. Providing information for sending notifications ................. 9
   4.4. Expiration of subscription ..................................... 9
   4.5. Common processing rules for subscriptions ...................... 10
      4.5.1. General processing rules for subscriptions ............... 10
      4.5.2. Processing rules for data the subscription applies to .......... 10
      4.5.3. Processing rules for `<Aggregation>` and `<Trigger>` .......... 11
      4.5.4. Processing rules for first notification and expiry of subscription .......... 11
      4.5.5. Processing rules when the access and privacy policies forbid subscription ........ 12
   4.6. SelectType for subscription objects ............................ 13
   4.7. Support for `<Subscription>` conditioned by `<TestItem>` .......... 13
5. Notifications ................................................................ 15
   5.1. `<Notify>` element ........................................... 15
   5.2. `<Notification>` element ....................................... 15
   5.3. `<NotifyResponse>` element .................................... 16
   5.4. Processing rules for notifications ............................... 16
6. Subscription and Notification Examples ...................................... 18
   6.1. Piggy backing a subscription to Query .......................... 18
   6.2. Creating Subscription Object .................................... 18
7. Checklist for Service Specifications ....................................... 20
8. Schemata .................................................................... 22
   8.1. Summary of DST Reference Model with Subscriptions and Notifications ................. 22
   8.2. Subscriptions Utility Schema Summary .......................... 24
   8.3. Schema for DST Reference Model with Subscriptions and Notifications .................. 25
   8.4. Subscriptions Utility Schema .................................... 29
References ...................................................................... 31
1. Introduction

This specification provides protocols for subscription and notification. A subscription is a mechanism by which a WSC can register to receive notifications from a WSP when some data changes or some event happens.

Since there is usually data involved, it is common that data services, based on Data Services Template [LibertyDST], will incorporate subscription features. A fair amount of this specification is dedicated to these situations, including subscription as a side effect of query or create, and subscription by explicit manipulation of subscription objects, using a DST derived interface.

However, subscriptions can profitably be employed even outside data services and there is no need to base a service on DST for it to use subscriptions. People Service [LibertyPeopleService] illustrates this approach. In such case, the service in question is responsible for providing the methods for subscription and subscription management, while using <Subscription> element as defined in this document.

1.1. Notation

When capitalized, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this specification are to be interpreted as described in [RFC2119]. When these words are not capitalized, they are meant in their natural-language sense.

Definitions for Liberty-specific terms can be found in [LibertyGlossary].

For better readability, the XML schema [Schema1] is described using schema grammar notation which is explained in [LibertyDST], Section 1.3 "Schema Grammar Formalism".

1.2. Liberty Considerations

This specification contains enumerations of values that are centrally administered by the Liberty Alliance Project. Although this document may contain an initial enumeration of approved values, implementors of the specification MUST implement the list of values whose location is currently specified in [LibertyReg] according to any relevant processing rules in both this specification and [LibertyReg].

1.3. Namespaces

The namespaces described in table 1 are used.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>URI</th>
<th>Description</th>
</tr>
</thead>
</table>
1.4. Applying Subscriptions to DST Based Services

While Subscriptions and Notifications can be adopted by any ID-WSF based service, if these features are to be adopted by a DST based service it SHOULD be done on basis of the extended reference model described in Section 8.3, i.e., this model should replace the model specified in [LibertyDST] Section 11.3 "DST Reference Model Schema".

Specifically, the service is expected to provide definitions for

1. `<Notify>` and `<NotifyResponse>`. In particular, the definition of `<Notify>` will depend on the data model and schema adopted by the service.

2. `<Subscription>`, to incorporate the aspects of service dependent query language.

3. An interface for manipulation of subscription objects. In DST model, the regular "CRUD" interface is used with the special object type `"Subscription"`. A service that is not otherwise DST based may wish to support DST interface just for subscription object manipulation. Providing this facility is optional for a service specification.

4. A means for establishment of subscriptions as side effect ("piggy backed") of other operations. Providing this facility is optional for a service specification. An example of how this could be accomplished as side effect of `<Create>` is provided in Section 3.2.
2. General Rules Regarding Subscriptions and Notifications

2.1. Second level <Status> codes for Subscriptions

Following second level status codes are defined for subscriptions:

- EmbeddedSubscriptionsNotSupported
- InvalidSubscriptionID
- MissingSubscriptionID

If a request or notification fails for some reason, the ref XML attribute of the <Status> element SHOULD contain the value of the itemID XML attribute of the offending element in the request message. Subscription and notifications messages use subscriptionID XML attributes instead of itemID XML attributes and those should be used when reporting failure statuses related to the subelements of subscription and notification messages. When the offending element does not have the itemID or subscriptionID XML attribute, the reference SHOULD be made using the value of the id XML attribute, if that is present.

If it is not possible to refer to the offending element (as it has no id, itemID, or subscriptionID XML attribute) the reference SHOULD be made to the ancestor element having a proper identifier XML attribute closest to the offending element.

Since both itemID and subscriptionID can be used to refer to failed element, the two IDs form one namespace. Care should be taken to avoid id values that would create ambiguity.

2.2. Discovery Option Keywords for Indicating Lack of Subscription Support

A WSP MAY register following discovery option keywords to indicate that it does not support certain types of subscription manipulations:

- urn:liberty:subs:noSubscribe
- urn:liberty:subs:noQuerySubscriptions

2.3. CRUD Manipulation using Object Type "_Subscription"

Service specifications that support subscriptions must use object type "_Subscription" to designate them.

As a service may support different types of objects, the SelectType MUST be defined so that it supports all different types of objects supported by the service, including "_Subscription".

If a service supports subscriptions, the SelectType MUST be specified so that it can carry strings containing XPath expressions. If the same service type supports objects, which do not use XPath but e.g., own special element structure, the SelectType MUST still make it possible to carry just strings, this might require specifying mixed="true", but a service type MUST NOT use real mixed type and have strings and elements at the same time, so either strings or subelements are allowed, but not both at the same time.

2.4. No itemIDRef for subscription related <ItemData>

The <ItemData> elements returning changed expiration times for subscriptions created based on the request message MUST NOT contain any itemIDRef XML attribute. They contain <Subscription> elements, which carry subscriptionID XML attributes (see Section 4).
3. Piggy Backing Subscriptions to DST Operations

N.B. Subscription to <Delete> is generally not meaningful and is not discussed here.

3.1. <Query> with <Subscription>

While querying data, it is possible to simultaneously subscribe to future changes of that data by including <Subscription> elements, see Section 4, inside the <Query>. These <Subscription> elements MUST refer to the <QueryItem> elements using <RefItem> elements to indicate that a WSC wants to subscribe to the same data it is querying. The <Subscription> elements MAY also have their own <ResultQuery> elements to define additional data a WSC wants to subscribe to. A service specification and a WSP MAY specify additional restrictions on how subscriptions are supported inside queries, or that they are not supported at all.

```
%QueryType: base(dst:RequestType)
subsref:TestItem*
subsref:QueryItem*
subsref:Subscription*
;
```

Figure 1. Definition of <Query> that supports piggy backed <Subscription>.

3.2. <Create> with <Subscription>

A <Create> element may also contain one or more <Subscription> elements e.g., to subscribe to future changes of the data just created (see Section 4).

```
%CreateType: base(dst:RequestType)
subsref:Subscription*
subsref:CreateItem+
subsref:ResultQuery*
;
```

Figure 2. Definition of <Create> that supports piggy backed <Subscription>.

The <CreateResponse> element contains in addition to <Status> element possible <ItemData> elements, which carry requested data related to the data just created. <ItemData> elements may also carry information about subscriptions, when a WSP changed or added the expiration time. For example, returned data could include a unique ID assigned to the data object just created.

3.3. <Modify> with <Subscription>

A <Modify> may contain <Subscription> element(s), when a WSC wants to subscribe to the data it is modifying. These <Subscription> elements MUST refer to the <ModifyItem> elements using <RefItem> element(s). The <Subscription> elements MAY also have own <ResultQuery> element(s) to define additional data a WSC wants to subscribe to. See Section 4 for more information. A service specification and a WSP MAY set additional restrictions, how subscriptions are supported inside modification requests, if the support is allowed at all.

A <ModifyResponse> may contain <ItemData> element(s). The elements can contain either data requested with <ResultQuery> elements or <Subscription> elements, when a WSP has modified the expiration time.

```
%ModifyType: base(dst:RequestType)
subsref:Subscription*
subsref:ModifyItem+
subsref:ResultQuery*
;
```

Figure 3. Definition of <Modify> that supports piggy backed <Subscription>.
4. Subscriptions

The subscriptions are a mechanism through which WSCs can request for notifications when specified event happens. The basic case is subscribing to change notifications to get updates when the data hosted by a data service related to a Principal changes. A WSC may subscribe to change notifications even before the data exists. For example, a WSC may want to know when a Principal adds an email address to her profile. The change of data is not the only possible reason for a notification, there can be service specific triggers for notifications, e.g., periodic notifications containing current values and notifications after a Principal switches on her terminal.

As the notifications do not only reveal the data they are carrying, but also that certain thing has just happened, WSPs must be very careful to make sure they honor the privacy of the Principals.

This document specifies one objectType, the "_Subscription". These can be accessed and manipulated like any other objects; they can be created, deleted, modified and queried. The difference to other object types is that "_Subscription" objects can be created also otherwise than with normal <Create>. <Subscription> element can be embedded in other request types to make it easier to subscribe to the data accessed with those requests. For example, a WSC may subscribe to the data it just modified with a <Modify>. This can be done by adding a <Subscription> element into the <Modify> request without a need to make a separate <Create> request to create a "_Subscription" object that way.

When subscriptions are supported in addition to creating them, deleting subscriptions with <Delete> MUST be supported. Renewing subscriptions by modifying the expiration time (expires XML attribute) using <Modify> SHOULD also be supported and modifying other parameters of subscriptions MAY be supported.

Notifications are carried inside <Notify> elements. The notifications are specified in Section 5.

4.1. <Subscription> element

The <Subscription> element contains all the parameters for a subscription. It defines, what data a WSC wants to have, where it should be sent, when a subscription expires, which events should trigger notifications, etc.

%SubscriptionType:
subs:RefItem*%subs:Extensio
@subscriptionID -> %lu:IDType@notifyToRef -> %xs:anyURI # Points to EPR in SOAP header@adminNotifyToRef? -> %xs:anyURI # Points to EPR in SOAP header@starts? -> %xs:dateTime@expires? -> %xs:dateTime@id? -> %xs:ID@includeData?: enum(Yes No YesWithCommonAttributes );

RefItem? -> %subs:RefItemType%RefItemType:
@subscriptionID? -> %lu:IDType@lu:itemIDRef # ref to TestItem, QueryItem or ResultQuery;

Subscription? -> %subsref:SubscriptionType%SubscriptionType: base(subs:SubscriptionType)subsref:ResultQuery*%subsref:Extensio
Aggregation? -> %subsref:AggregationTypeTrigger? -> %subsref:TriggerType;

Figure 4. Utility schema for Subscription

Figure 5. Reference Model definition of Subscription
4.2. Selecting data a subscription applies to

The first parameter inside the `<Subscription>` element is the `<ResultQuery>` element. This is the basic data selection element used in multiple places. It defines what data a notification should return. The use of the `<ResultQuery>` element inside the `<Subscription>` element might be a bit different than its use when querying and modifying. The specifications for services MUST specify possible differences. Different parameters of the `<ResultQuery>` element are described together with processing rules in [LibertyDST] Section 4 "Querying Data". There can be more than one `<ResultQuery>` element inside a `<Subscription>` element.

The `<RefItem>` element is used when a WSC wants to subscribe to the data it is accessing with the `RequestElement`. Finally, a `<Trigger>` element can be used to specify arbitrary conditions for triggering notifications.

Normally a notification is triggered when the data addressed by the `<ResultQuery>` or `<RefItem>` element has changed. There can be also other reasons that trigger notifications. The `<Trigger>` element contains those triggers.

The `<Trigger>` element is of type `TriggerType`, which MUST be defined by the services schema; the service specification MUST define semantics and values for this parameter. When the `<Trigger>` element is not used, a WSC is requesting normal change notifications unless otherwise specified by a service specification.

4.3. Providing information for sending notifications

The XML attribute `notifyToRef` contains a reference to an endpoint object, defined in the SOAP headers of the message, which indicates where and how (e.g., using which security mechanism and credentials or tokens) the notification must be sent. The `notifyToRef` and `adminNotifyToRef` design pattern and the associated end point objects are further described in [LibertySOAPBinding].

If the `adminNotifyToRef` XML attribute is not specified, the subscription end notifications are sent to end point indicated by the `notifyToRef` XML attribute. The purpose of the `adminNotifyToRef` XML attribute is to make it possible to receive notifications in one point and manage changes to subscriptions in another point.

There can be different types of notifications. For example, a notification can be sent immediately, or multiple notifications could be sent in a bigger batch. The element `<Aggregation>` defines, what type of notifications a WSC is requesting. The element `<Aggregation>` further describes, in a service specification dependent way, how the notifications are to be batched. It is of type `AggregationType`, which MUST be specified, including the detailed semantics and allowed values, by the service specification.

Usually a notification contains data related to a resource. Sometimes a notification could be used to indicate that an event related to a resource has happened, e.g., the data addressed by the `<ResultQuery>` element has changed, without reporting the changed data. The XML attribute `includeData` defines whether or not the data of the changed object be included in the notification messages. Possible values are `Yes` (data is returned), `No` (no data is returned), and `YesWithCommonAttributes` (the data is returned with the common XML attributes). A service specification SHOULD specify a default value. It should be noted that sending just a change notification without any actual data usually has less security and privacy issues compared to cases, when the data is also included in a notification message.

4.4. Expiration of subscription

A subscription is not valid forever. The `starts` XML attribute defines the time after which a subscription is valid and notifications can be sent if the triggering event occurs. The `starts` XML attribute MUST be used only when a subscription is not valid immediately after processing the request. The `expires` XML attribute defines the time when a subscription expires, if not renewed before that time.
If credentials needed for subscription expire earlier than a subscription, and a WSC does not provide new credentials before they expire, the subscription MUST expire.

### 4.5. Common processing rules for subscriptions

When subscriptions are requested by a WSC, the following processing rules MUST be obeyed (note these rules are valid regardless the way a subscription is requested).

A subscription is one entity, which either succeeds or fails. A subscription is identified with a subscriptionID.

#### 4.5.1. General processing rules for subscriptions

1. If a WSP fails to process the parameters of a subscription properly according to the specified rules, it MUST NOT accept that subscription and SHOULD use appropriate second level status code to indicate the reason. Note that one <Subscription> element in a request message may specify more than one subscription since <RefItem> elements may have their own subscriptionID XML attributes. So one <Subscription> element may contain subscriptions which succeed and subscriptions which fail. Failure of even one subscription SHOULD cause error response, unless the service specification specifies rules for partial success.

2. When subscriptions are created within <Query> or <Modify> or within <Create> so that they are direct child elements of the <Create> (referring to <CreateItem> elements), the failure to process subscription or rejecting those for other reasons, e.g., policies, is not considered as a failure of a <Query>, <Modify> or <Create> request. The normal <Query>, <Modify>, or <Create> parameters inside <QueryItem>, <ModifyItem>, or <CreateItem>, elements respectively, MUST be processed normally even if a subscription referring to those fails unless otherwise stated by a service specification. If a subscription is not accepted, a WSP MUST indicate this back to a WSC. For example, if a WSP does not support <Subscription> elements embedded as a direct child of a <Query>, a <Modify>, or a <Create> element and it receives such, it MUST use second level status code EmbeddedSubscriptionsNotSupported to indicate this. If processing of an embedded <Subscription> element fails, proper second level status code MUST be returned and the failed <Subscription> element MUST be referred to using the subscriptionID as the value of the ref XML attribute of the <Status> element. As failing embedded subscription does not cause failure of a request message a WSC MUST check the return second level status elements to find out, where those subscriptions accepted by a WSP or not.

3. When a new subscription is created the way data objects are normally created (i.e. within <NewData> of a <CreateItem>), the normal processing rules MUST be applied with the exception that this specification gives some object type specific processing rules and more detailed status codes to be used when applicable instead of generic InvalidData. When a WSP does not support subscriptions and a WSC tries to create one in the way data objects are created, it should return second level status code UnsupportedObjectType, when subscriptions are allowed for the service type, but not supported by a WSP, and InvalidObjectType, when they are not allowed for the service type.

4. The values of the subscriptionID XML attributes are WSC specific. When a new subscription is created, it MUST use a subscriptionID different from any other subscription the same WSC has at the same WSP. If a WSC tries to create a new subscription which has conflicting subscriptionID value, a WSP MUST reject that and it SHOULD use second level status code InvalidSubscriptionID.

5. An implementation MAY decompose a composite subscription object into unit subscriptions. For example, if a subscription object has multiple <RefItem> elements with different subscriptionID XML attributes, this is interpreted to create multiple logical subscriptions. An implementation may indeed choose to handle them as separate subscriptions. While implementation MUST support creation of composite subscriptions, it NEED NOT support composite subscriptions on <Query>, <Modify>, and <Delete> interfaces involving objects of type ".Subscription".
4.5.2. Processing rules for data the subscription applies to

A WSC must specify in a subscription the data the subscription applies to.

1. When a <Subscription> element contains an <ResultQuery> element, a WSP MUST process its content in a similar fashion as it processes the same parameters in a case of a normal query taking into account that no data is returned immediately. A WSP MUST support requested objectType and <Select>. If a WSP does not support sorting and it is requested by a WSC, a WSP SHOULD still accept the subscriptions and return data unsorted in notifications. The changedSince XML attribute MUST be ignored, if present. When notifications are expected to contain only the changed data, a WSC MAY use <ChangeFormat> to indicate formats it supports. Note that with subscriptions the <ChangeFormat> is used without having the changedSince XML attribute (required in regular queries). The predefined XML attribute can be used instead of other parameters. See [LibertyDST], Section 3.7 "Selection" and Section 4.4 "Processing Rules for Queries" for more details and proper status codes.

2. When a <RefItem> element is included in a subscription, it MUST contain an itemIDRef XML attribute. The value of this XML attribute MUST be the same as the value of an itemID XML attribute of a <QueryItem>, a <CreateItem> or a <ModifyItem> depending on the message. This creates a subscription to all of the data manipulated in the referenced element.

3. If the value of the itemIDRef XML attribute does not match to any relevant itemID, the subscription MUST NOT be accepted and second level status code InvalidItemIDRef SHOULD be used to indicate the reason.

4. If a <RefItem> element contains a subscriptionID XML attribute, and it has different value than the subscriptionID XML attribute of the <Subscription> element, the <RefItem> element defines a new subscription which inherits other parameters except <ResultQuery> elements, subscriptionID XML attribute and possible other <RefItem> elements from the <Subscription> element in which the <RefItem> element is contained. Each <RefItem> that has subscriptionID XML attribute creates a new independent subscription. If multiple <RefItem> elements have the same value of the subscriptionID, they all form one subscription together and that subscription has multiple sets of selection parameters. If data selected by any of the sets is changed, a notification is sent.

5. A <Subscription> element may contain any number of <ResultQuery>, <RefItem>, and <Trigger> elements. If none of the elements <ResultQuery>, <RefItem> or <Trigger> are present, the processing of the <Subscription> element MUST fail unless the service specification has defined, what this kind of a case means, e.g., some default values are defined for parameters and those are used or a WSC subscribes to the whole resource. When the processing of a <Subscription> element fails due to not having <ResultQuery>, <RefItem> or <Triggers> present, the second level status code MissingSelect SHOULD be used to indicate this.

4.5.3. Processing rules for <Aggregation> and <Trigger>

1. A WSP MUST follow the processing rules defined in the service specification for the elements <Aggregation> and <Trigger>. If the use of these elements is not specified for the service or specified, but not supported by a WSP, and either of both of them are included in a <Subscription> element in a <Subscribe> request, the processing of the <Subscription> MUST fail and a second level status code indicating this SHOULD be used, either AggregationNotSupported or TriggerNotSupported.

2. If a WSP does support aggregation, but not the type of <Aggregation> a WSC requests, the processing of the <Subscription> MUST fail and a second level status code RequestedAggregationNotSupported indicating this SHOULD be used in addition to the top level status code. Similarly if a WSP does support triggers, but not the type of a <Trigger> a WSC requests, the processing of the <Subscription> MUST fail and a second level status code RequestedTriggerNotSupported indicating this SHOULD be used in addition to the top level status code.
4.5.4. Processing rules for first notification and expiry of subscription

A WSC may request when the first notification may be sent and when a subscription should expire.

1. If a `<Subscription>` element contains a `starts` XML attribute, subscription MUST be valid only after the time defined. If the `starts` XML attribute is omitted, the subscription MUST be valid immediately after processing the request. Also, if the time specified by the `starts` XML attribute is in the past, then that subscription, if accepted by a WSP, MUST be valid immediately after processing the request.

2. The time specified by the `expires` XML attribute MUST be the same time or a later time than the time specified by the `starts` XML attribute in the same `<Subscription>` element. It also MUST be later than the current time. If either of the checks is not passed, then the processing of the `<Subscription>` MUST fail and a second level status code `InvalidExpires` indicating this should be used in addition to the top level status code.

3. A WSP MAY change the time when a subscription expires from the expiration time requested by a WSC with the `expires` XML attribute. A WSP MAY shorten the expiration time, but it MUST NOT make the expiration time longer. If no `expires` XML attribute is included in a `<Subscription>` element in a request from a WSC, a WSP MUST decide the expiration time for the subscription, if expiration times are required either by the service specification or the WSP. A WSP MUST return the expiration time in the response message, if it is changed compared to what a WSC requested. This information is returned by returning a `<Subscription>` element with XML attributes `subscriptionID` and `expires` inside `<Data>` element in case of a `<QueryResponse>` and `<ItemData>` in case of a `<CreateResponse>` and `<ModifyResponse>`. That `<Data>` or `<ItemData>` element MUST NOT contain any other data than `<Subscription>` elements created based on one `<Subscription>` element or, when normal data object creation method has been used. `<Subscription>` elements created with one `<CreateItem>` element. The `<Data>` or `<ItemData>` element SHOULD NOT contain any `itemIDRef` XML attributes, the matching is done based on the `subscriptionID` XML attributes carried inside `<Subscription>` elements.

4. If a WSC wants to renew an existing subscription before it has ended, it MUST modify that subscription and give a new value for the `expires` XML attribute of that subscription. A WSP MAY modify the new value in the same way as it MAY modify the proposed value for a new subscription.

5. There is one special case, when using subscriptions expiration. When the `starts` and the `expires` XML attributes have exactly same values, the meaning is that a notification MUST be sent exactly at that time whether some event (e.g., data change) has happened or not. A WSC wants to get current values of the data (e.g., location) exactly at that time, even if the values have stayed the same for a long time (a Principal has not moved).

4.5.5. Processing rules when the access and privacy policies forbid subscription

The access and privacy policies specified by the resource owner may not allow a WSC to subscribe to the data of a resource or to some events related to a resource.

1. When a WSP processes a `<Subscription>` element, it MUST check whether the resource owner (the Principal, for example) has given consent to return the requested data and the fact that an event or data change has happened in notification messages. To be able to check WSC-specific access rights, the WSP MUST authenticate the WSC (see [LibertySecMech]). The WSP MUST also check that any usage directive given in the request is acceptable based on the usage directives defined by the resource owner (see [LibertySOAPBinding]). If either check fails, the WSP MUST NOT accept the subscription and the processing of that `<Subscription>` MUST fail. The WSP MAY try to get consent from the Principal while processing the request, perhaps by using an interaction service (see [LibertyInteract]). A WSP might check the access rights and policies in usage directives at a higher level, before getting to DST processing and MAY, in this case, just return an ID-* Fault Message [LibertySOAPBinding] without processing the `RequestElement` element at all, if the requesting WSC is not allowed to access the data in question.
2. Note that there can be consent for subscribing to some data element, but not its XML attributes. A Principal might not want to release the modifier XML attribute, if she does not want to reveal information about which services she uses. If a WSC is not allowed to get all the data, but some data it wants, a WSP SHOULD accept the subscription, but it MAY also reject it. If a subscription is accepted, the data for which there is no consent from the Principal MUST be handled as if there was no data. Also that data or the fact that data has changed MUST NOT be included in the notification messages sent later on.

3. If a WSC has made a subscription and included the usage directive it has promised to obey and later wants to change the usage directive, it MUST cancel the subscription and make a new subscription with the new value for the usage directive.

4.6. SelectType for subscription objects

N.B. This subsection is about selecting the wanted subscription objects when deleting and modifying them, not about a subscription selecting the right data for notifications. When a WSC wants to access existing subscription objects after they have been created, it must be able to select the right ones. XPath is used to select the subscription objects.

The minimum a WSP MUST support is /ns:Subscription[@ns:subscriptionID="xx"] so that a WSC can delete an existing subscription using <Delete>. Of course the objectType XML attribute must have the value "_Subscription". Just by setting the objectType XML attribute to "_Subscription" a WSC can delete all subscription it has related to a resource.

A WSP SHOULD also support /ns:Subscription[@ns:subscriptionID="xx"]/@ns:expires to make it possible to renew a subscriptions before it expires using the <Modify>.

A WSP MAY also support:

/ns:Subscription[@ns:subscriptionID="xx"]/ns:notifyToRef
/ns:Subscription[@ns:subscriptionID="xx"]/ns:adminNotifyToRef

to make it possible to change endpoints and related information.

A WSP MUST support abbreviated XPath, as described in [XPATH] Section 2.5.

A WSP MAY also support full XPath to make it possible to modify all the parameters of a subscription without a need to rewrite those parameters, which do not change, but a subscription can be updated by selecting it using ~/ns:Subscription[@ns:subscriptionID]~ and rewriting the whole subscription

4.7. Support for <Subscription> conditioned by <TestItem>

A WSC can subscribe to be notified if the results of a test change. For example, if the original result of a test was true, the WSC can ask to be notified when the result becomes false and vice versa.

The WSC indicates that it is subscribing to the test results by specifying itemIDRef XML attribute that references the appropriate <TestItem> element. The result is reported via <TestResult> in the <Notification>.

1. A service specification MAY restrict, or forbid, use of <TestItem> in conjunction with <Subscription>, if use of <TestItem> is fully supported, the WSP MAY register the discovery option keyword

urn:liberty:subs:contingentSubscription
2. A `<Subscription>` that references `<TestItem>` MUST NOT have `<Trigger>`. The only valid triggering condition is "on change", which is implied, thus no `<Trigger>` element is necessary.

3. If the `itemIDRef` attribute does not match a `<TestItem>` then the WSP MUST stop processing the `<Subscription>` and return a second level status code `NoSuchTest`.

4. If `<Subscription>` has a `itemIDRef` XML attribute, the WSP MUST detect changes to the result of evaluation of the `<TestItem>` referenced by the `itemIDRef` and send notifications when they occur.

5. The scope of the `itemIDRef` is one `<Query>`, `<Create>`, or `<Modify>`. `itemIDRef` MUST NOT refer to `itemID` in another top level element. The `itemID` XML attributes of `<TestItem>` elements MUST be unique within one `<Query>`, `<Create>`, or `<Modify>` element in the request. The `<TestItem>`, `<ResultQuery>`, and `<QueryItem>` share same `itemID` space.
5. Notifications

When a WSC has subscribed to some data or event, a WSP will send notifications when the subscribed data changes or the event happens. A notification can also be sent when a subscription expires or is changed by a WSP (e.g., it shortens the expiration time).

5.1. <Notify> element

Notifications are carried by <Notify> elements. One <Notify> element may carry one or more <Notification> elements. Otherwise the <Notify> element just has the normal id and timestamp XML attributes.

```xml
%@NotifyAttributeGroup:
  @timeStamp? -> %xs:dateTime
;,

%NotificationType:
  lu:TestResult*
  @id? -> %xs:ID
  @subscriptionID -> %lu:IDType
  @expires? -> %xs:dateTime
  @endReason? -> %xs:anyURI
;,

%NotifyResponseType: base(lu:ResponseType);

Figure 6. Utility schema for Notify
```

```xml
%NotifyType: base(dst:RequestType)
subsr:Notification*
@subsr:NotifyAttributeGroup
;,

Notification -> %subsr:NotificationType
%NotificationType: base(subs:NotificationType)
subsr:ItemData*
;,

%NotifyResponseType: base(subs:NotifyResponseType);

Figure 7. Reference Model definition of Notify
```

5.2. <Notification> element

The main content of the <Notification> element is the <ItemData> element, which contains the data the notification carries, e.g., the current location or the changed home address. In a case of a change notification the same formats as in responses to queries for changed data are used.

The <ItemData> element may also contain some other type of data indicating what kind of an event has happened. The whole <ItemData> element might not be used at all as it is possible to subscribe to notifications to indicate that an event has happened, e.g., data has changed without having the data in a notification message. The subscriptionID indicates what data has changed. For privacy reasons this is the recommended alternative in many cases.

In addition to the <ItemData> element(s) the <Notification> element has a number of XML attributes. The subscriptionID XML attributes identifies the subscription based on which a notification is sent. So one <Notification> element carries information only related to one <Subscription>. A <Notify> element may carry multiple <Notification> elements.
One `<ItemData>` element MUST NOT contain more data that address by one `<ResultQuery>` or `<RefItem>` of the subscription.

The `expires` XML attribute is used to indicate in a notification message the time, when the subscription will expire. In an administrative notification the `endReason` XML attribute can be used to indicate the reason for the end of the subscription. This might give some indication to a WSC that a WSP is having some problems or whether it makes sense or not for a WSC to try to make the subscription again. The `endReason` XML attribute is not used in normal notifications, only when administrative notifications are used to notify that a subscription has ended. Possible values for the `endReason` XML attribute include:

- `urn:liberty:subs:endreason:unspecified` - The real reason is unspecified.
- `urn:liberty:subs:endreason:wscnotacknowledging` - A WSP cancels the subscription as it have not got acknowledgments from a WSC to the notification messages.
- `urn:liberty:subs:endreason:resourcedeleted` - The resource has been deleted, so there is no data anymore available.
- `urn:liberty:subs:endreason:expired` - The subscription has expired, either a WSC did not renew it in time or a WSP changed the expiration time.
- `urn:liberty:subs:endreason:credentialsexpired` - The credentials given for sending notifications have expired and so a WSP is not capable of sending notifications anymore. This notification might have to be sent just before the credentials are about to expire as otherwise even this notification can not be sent.

A WSP must be careful not to compromise the privacy of a Principal, when sending the reason codes for ending a subscription.

### 5.3. `<NotifyResponse>` element

Notifications are acknowledged using the `<NotifyResponse>` element. It contains only the `<Status>` element. A service specification MUST specify whether notifications acknowledgments are used or not or whether it is an implementation or deployment specific decision.

### 5.4. Processing rules for notifications

The common processing rules specified in [LibertyDST], Section 3 "Message Interface", MUST also be followed.

1. A WSP MUST send a notification message to a WSC which has made a subscription, when an event defined by the parameters of that subscription happens. When sending these normal notification message to a WSC, a WSP MUST use the information provided in the XML attribute `notifyToRef` element (endpoint, security mechanism, and credentials or tokens).

2. When a subscription becomes invalid or has been changed by a WSP somehow, a WSP MUST send a notification to indicate this, if administrative notifications about subscriptions are used. When a WSP is sending a notification about expiration or change of a subscription, it MUST use the information provided in the XML attribute `adminNotifyToRef` (endpoint, security mechanism, and credentials or tokens). If the `adminNotifyToRef` XML attribute is not specified the `notifyToRef` element is used instead.

3. If the receiving WSC can not successfully process one of the `<Notification>` elements inside one `<Notify>` element, it SHOULD process normally the rest of the `<Notification>` elements and try to achieve a partial success. A WSC MUST support multiple `<Notification>` elements inside one `<Notify>` element.
4. `<Notification>` element inside a notification message MUST have a `subscriptionID` XML attribute to identify the subscription based on which the notification message is sent. If the `subscriptionID` XML attribute is missing, the processing of that `<Notification>` element MUST fail and a second level status code `MissingSubscriptionID` SHOULD be used in addition to a top level status code. If a WSC does not recognize the value of a `subscriptionID` XML attribute, the processing of that `<Notification>` element MUST fail and a second level status code `InvalidSubscriptionID` SHOULD be used in addition to a top level status code.

5. A `<Notification>` element inside a notification message MUST have the `expires` XML attribute, if subscription expiration is used. When a WSC receiving a notification knows that the `expires` XML attribute should have been used, but it is not, it SHOULD use the second level status code `MissingExpiration`. Irrespective of reporting the missing `expires`, the WSC MAY decide whether it considers this a failure or not.

6. One `<Notification>` element MUST NOT contain both the data subscribed and information about the change of a subscription. The only exception is the expiration time. If a WSP changes the expiration time, an administrative notification is sent, if used, but the new expiration time is also included in the normal notifications.

7. If a `<Notification>` element is supposed to contain data about a resource (i.e., the `includeData` XML attribute of a subscription has either the value `Yes` or `YesWithCommonAttributes`), the `<ItemData>` element MUST be used in a `<Notification>` element. The content of a `<ItemData>` element MUST be according to the parameters of the related subscription, especially `<ResultQuery>` or `<RefItem>`, and the related event, which has caused this `<Notification>` element to be sent inside a notification message. In case of a change notification, the same formatting rules for the content as in case of a query for changes MUST be followed (see [LibertyDST], Section 4 "Querying Data"). A WSP MUST NOT include any data, which the WSC is not allowed to get based on access rights and privacy policies defined by the resource owner. If an `<ItemData>` element should have been included in a `<Notification>` element, but it is missing, the processing of the `<Notification>` element MUST fail and a second level status code `MissingDataElement` SHOULD be used in addition to the top level status code.

8. For change notification a `changeFormat` XML attribute MUST be added for a `<ItemData>` element to indicate the format used to shown the changes, if a service specification has not mandated only one specific format to be used for this.

9. If the data inside an `<ItemData>` element is invalid, the processing of the `<Notification>` element MUST fail and a second level status code `InvalidData` SHOULD be used in addition to the top level status code. A WSC MUST accept all the data, which can be considered as possible normal extension, if extensions are allowed for a service based on the service specification.

10. If a `<Notification>` element has the `endReason` XML attribute, the notification is expected to indicate the end of the subscription and all other content of the `<Notification>` element except the `subscriptionID` XML attribute MUST be ignored unless some service specific extensions needed in this kind of cases have been specified. The `endReason` XML attribute MUST have a value specified in this document or valid service or implementation specific value. A WSP MUST be careful not to use any value which might compromise the privacy of a Principal.

11. A WSP SHOULD resend a notification for which it does not get an acknowledgment in reasonable time, if acknowledgments are used. If a WSP does not get acknowledgments at all within its time and other limits, it MAY cancel the related subscription.
6. Subscription and Notification Examples

6.1. Piggy backing a subscription to Query

Consider a subscription to data that is queried

```xml
<Query>
  <QueryItem itemID="djkfgjkdf">
    <Select>/hp:HP/hp:AddressCard</Select>
  </QueryItem>
  <Subscription includeData="Yes" subscriptionID="tr578k-kydg4b" notifyToRef="#123">
    <RefItem itemIDRef="djkfgjkdf"/>
  </Subscription>
</Query>
```

Here we see itemIDRef referencing the `<QueryItem>` to define the data to be subscribed. The subscriber also allocates a subscriptionID and provides the end point to contact by way of notifyToRef XML attribute that references an endpoint in the SOAP headers (not shown).

This subscription could later generate following notification

```xml
<Notify>
  <Notification subscriptionID="tr578k-kydg4b">
    <ItemData>
      <hp:AddressCard id="9812">
        <hp:AddressType>urn:liberty:id-sis-hp:addrType:home</hp:AddressType>
        <hp:Address>
          <hp:C>us</hp:C>
        </hp:Address>
      </hp:AddressCard>
    </ItemData>
  </Notification>
</Notify>
```

The salient point to notice is that the `<Notification>` correlates to the subscription using subscriptionID XML attribute.

6.2. Creating Subscription Object

Consider

```xml
<Create>
  <CreateItem objectType="_Subscription" itemID="1">
    <NewData>
      <Subscription subscriptionID="subs123" notifyToRef="#1" includeData="1">
        <ResultQuery objectType="entry">
          <Select attributes="HELLO"></Select>
        </ResultQuery>
      </Subscription>
    </NewData>
  </CreateItem>
</Create>
```

The above example illustrates
a. Creating subscription by explicit creation of object of type "]_Subscription"

b. Defining notification data using `<ResultQuery>`

c. Creating a subscription to the data of `<CreateItem>` by referencing it using `<RefItem>`

d. Subscribing to the changes to the subscription itself.
7. Checklist for Service Specifications

1. Provide schema for `<Notify>`, `<NotifyResponse>`, and `<Subscription>` elements. If these are named differently, indicate the correspondence to the standard naming.

2. If service supports subscriptions, it will need to handle the object type "Subscription". If the AppDataType is defined using XML schema, this schema needs to make allowance, e.g., by using `<xs:choice>` construct, for `<Subscription>` elements by referencing the DST schema.

If service adopts the default definition of AppDataType, which uses mixed content model, then the default is that all string data belongs to service specified object types while any `<Subscription>` containers belong to object type "Subscription". Further, element of type AppDataType may only contain objects of one type.

3. Describe how SelectType applies to subscriptions.

In particular, if service supports subscriptions, it MUST provide a way to specify XPath expressions for querying them. The XPath expressions MAY be restricted to the subset described in Section 4.6. This MAY be specified by stating that "default restriction on XPaths for subscriptions applies".

4. Describe TriggerType or state that it is not used.

5. Describe AggregationType or state that it is not used.

6. Extension support.

   a. If TriggerType, or AggregationType is designated as unused by the service specification, then it MAY be used for extension, provided that the extension data is

      a. in URI format and use an assigned domain name as a component of the URI to ensure that extensions do no collide with each other.

      b. Namespace qualified XML document

7. Statement of how subscriptions can be established and manipulated

   a. Support CRUD manipulation of subscriptions as objectType "Subscription"

   b. Support subscribing in `<Query>`

   c. Support multiple `<Subscription>` elements in `<Query>`

   d. Support subscribing in `<Create>`

   e. Support multiple `<Subscription>` elements in `<Create>`

   f. Support subscribing in `<Modify>`

   g. Support multiple `<Subscription>` elements in `<Modify>`

8. Start of a subscription. Usually a subscription is valid after it has been created, but if supported, a WSC may request that a subscription is valid only after a specific time using the `starts` XML attribute. It MUST be specified here, if the `starts` XML attribute supported or not.

9. Subscription expiration. Usually subscriptions expire after a certain time, but a service specification may also specify e.g., that subscription expiration is not used and WSCs must cancel subscriptions after they are not needed. It MUST be specified here, do subscriptions expire or not, e.g., Subscription expiration MUST be used.
10. Support expires==starts. Is it allowed to specify same time both for the starts and the expires XML attribute to request one notification message at a specified time, e.g., same value MAY be used both for the starts and the expires XML attribute.

11. Support querying existing subscriptions. Some services or implementations may or may not support querying existing subscriptions. This should be stated here. E.g. MUST NOT be supported.

12. Are notifications acknowledged? Some services or implementations may or may not support acknowledging notifications using <NotifyResponse>. This should be stated here, e.g., Notifications MUST BE acknowledged.
8. Schemata

8.1. Summary of DST Reference Model with Subscriptions and Notifications

The reference model schema can be summarized as follows

```xml
<target(subsref, urn:liberty:subs:2006-02:ref)
import(dst, urn:liberty:dst:2006-02, liberty-idwsf-dst-v2.1.xsd)
import(subs, urn:liberty:subs:2006-02, liberty-idwsf-subsv1.0.xsd)
import(lu, urn:liberty:util:2005-11, liberty-idwsf-utility-v2.0.xsd)
```

#sec(methods)
```
Create -> %subsref:CreateType
CreateResponse -> %subsref:CreateResponseType
Query -> %subsref:QueryType
QueryResponse -> %subsref:QueryResponseType
Modify -> %subsref:ModifyType
ModifyResponse -> %subsref:ModifyResponseType
Delete -> %subsref:DeleteType
DeleteResponse -> %subsref:DeleteResponseType
```

#endsec(methods)

#sec(notifymethods)
```
Notify -> %subsref:NotifyType
NotifyResponse -> %subsref:NotifyResponseType
```

#endsec(notifymethods)

#sec(redefs)
```
%SelectType: base(xs:string) ;
%TestOpType: base(xs:string) ;
%SortType: base(xs:string) ;
%TriggerType: base(xs:string) ;
%AggregationType: base(xs:string) ;
%AppDataType: mixed(1)
subsref:Subscription*
```

#endsec(redefs)

# ========================================

#sec(create)
```
%CreateType: base(dst:RequestType)
subsref:Subscription*
subsref:CreateItem+
subsref:ResultQuery*
```

#endsec(create)

#sec(createaux)
```
CreateItem -> %subsref:CreateItemType
%CreateItemType:
subsref:NewData?
&@dst:CreateItemAttributeGroup
```

#endsec(createaux)

# ========================================

Liberty Alliance Project
Liberty Alliance Project:
Liberty ID-WSF Subscriptions and Notifications

#sec(query)
%QueryType: base(dst:RequestType)
subsref:TestItem*
subsref:QueryItem*
subsref:Subscription*
;
#endsec(query)

#sec(queryaux)
TestItem -> %subsref:TestItemType
%TestItemType: base(dst:TestItemType)
TestOp? -> %subsref:TestOpType
;
QueryItem -> %subsref:QueryItemType
%QueryItemType: base(subsref:ResultQueryType)
@dst:PaginationAttributeGroup
;
#endsec(queryaux)

#sec(queryresp)
%QueryResponseType: base(dst:DataResponseType)
lu:TestResult*
subsref:Data*
;
Data -> %subsref:DataType
%DataType: base(subsref:ItemDataType)
@dst:PaginationResponseAttributeGroup
;
#endsec(queryresp)

#sec(mod)
%ModifyType: base(dst:RequestType)
subsref:Subscription*
subsref:ModifyItem+
subsref:ResultQuery*
;
#endsec(mod)

#sec(modaux)
ModifyItem -> %subsref:ModifyItemType
%ModifyItemType:
subsref:Select?
subsref:NewData?
@dst:ModifyItemAttributeGroup
;
%ModifyResponseType: base(subsref:DataResponseType)
;
#endsec(modaux)

#sec(del)
%DeleteType: base(dst:RequestType)
subsref:DeleteItem+
;
DeleteItem -> %subsref:DeleteItemType
%DeleteItemType:
subsref:Select?
;
%DeleteResponseType: base(lu:ResponseType)
;
#endsec(del)
8.2. Subscriptions Utility Schema Summary

The utility schema can be summarized as follows:

target(subs, urn:liberty:subs:2006-02)
import(lu, urn:liberty:util:2005-11, liberty-idwsf-utility-v2.0.xsd)

@subscriptionID -> %lu:IDType
@notifyToRef -> %xs:anyURI # Points to EPR in SOAP header
@adminNotifyToRef? -> %xs:anyURI # Points to EPR in SOAP header
@starts? -> %xs:dateTime
@expires? -> %xs:dateTime
@id? -> %xs:ID
@includeData?: enum( Yes No YesWithCommonAttributes ) ;
The formal scheme for the reference model follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema

targetNamespace="urn:liberty:subs:2006-02:ref"
xmlns:subsref="urn:liberty:subs:2006-02:ref"
xmlns:dst="urn:liberty:dst:2006-02"
xmlns:subs="urn:liberty:subs:2006-02"
xmlns:lu="urn:liberty:util:2005-11"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
<xs:import namespace="urn:liberty:dst:2006-02"
schemaLocation="liberty-idwsf-dst-v2.1.xsd"/>
<xs:import namespace="urn:liberty:subs:2006-02"
schemaLocation="liberty-idwsf-subs-v1.0.xsd"/>
<xs:import namespace="urn:liberty:util:2005-11"
schemaLocation="liberty-idwsf-utility-v2.0.xsd"/>
<!--sec(methods)-->  
<xs:element name="Create" type="subsref:CreateType"/>
<xs:element name="CreateResponse" type="subsref:CreateResponseType"/>
<xs:element name="Query" type="subsref:QueryType"/>
<xs:element name="QueryResponse" type="subsref:QueryResponseType"/>
<xs:element name="Modify" type="subsref:ModifyType"/>
<xs:element name="ModifyResponse" type="subsref:ModifyResponseType"/>
<xs:element name="Delete" type="subsref:DeleteType"/>
<xs:element name="DeleteResponse" type="subsref:DeleteResponseType"/>
<!--endsec(methods)-->  
<!--sec(notifymethods)-->  
<xs:element name="Notify" type="subsref:NotifyType"/>
<xs:element name="NotifyResponse" type="subsref:NotifyResponseType"/>
<!--endsec(notifymethods)-->  
<!--sec(redefs)-->  
<xs:complexType name="SelectType">
<xs:simpleContent>
<xs:extension base="xs:string"/>
</xs:simpleContent>
</xs:complexType>
<!--endsec(redefs)-->  
</xs:schema>
```

8.3. Schema for DST Reference Model with Subscriptions and Notifications

The formal scheme for the reference model follows.
<xs:complexType name="TestOpType">
  <xs:simpleContent>
    <xs:extension base="xs:string"/>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="SortType">
  <xs:simpleContent>
    <xs:extension base="xs:string"/>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="TriggerType">
  <xs:simpleContent>
    <xs:extension base="xs:string"/>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="AggregationType">
  <xs:simpleContent>
    <xs:extension base="xs:string"/>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="AppDataType">
  <xs:complexContent>
    <xs:extension base="dst:RequestType">
      <xs:sequence>
        <xs:element ref="subsref:Subscription" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="subsref:CreateItem" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="subsref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="CreateItemType">
  <xs:sequence>
    <xs:element ref="subsref:NewData" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attributeGroup ref="dst:CreateItemAttributeGroup"/>
</xs:complexType>

<xs:complexType name="CreateResponseType">
  <xs:complexContent>
    <xs:extension base="subsref:DataResponseType">
      <xs:sequence>
        <xs:element ref="subsref:ItemData" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="DataResponseType">
  <xs:complexContent>
    <xs:extension base="dst:DataResponseBaseType">
      <xs:sequence>
        <xs:element ref="subsref:ItemData" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="QueryType">
  <xs:complexContent>
    <xs:extension base="dst:DataResponseBaseType">
      <xs:sequence>
        <xs:element ref="subsref:ItemData" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexContent>
<xs:extension base="dst:RequestType">
<xs:sequence>
<xs:element ref="subsref:TestItem" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="subsref:QueryItem" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="subsref:Subscription" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<!--endsec(query)-->  
<!--sec(queryaux)-->  
<xs:element name="TestItem" type="subsref:TestItemType"/>
<xs:complexType name="TestItemType">
<xs:complexContent>
<xs:extension base="dst:TestItemBaseType">
<xs:sequence>
<xs:element name="TestOp" minOccurs="0" maxOccurs="1" type="subsref:TestOpType"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:element name="QueryItem" type="subsref:QueryItemType"/>
<xs:complexType name="QueryItemType">
<xs:complexContent>
<xs:extension base="subsref:ResultQueryType">
<xs:attributeGroup ref="dst:PaginationAttributeGroup"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<!--endsec(queryaux)-->  
<!--sec(queryresp)-->  
<xs:complexType name="QueryResponseType">
<xs:complexContent>
<xs:extension base="dst:DataResponseBaseType">
<xs:sequence>
<xs:element ref="lu:TestResult" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="subsref:Data" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:element name="Data" type="subsref:DataType"/>
<xs:complexType name="DataType">
<xs:complexContent>
<xs:extension base="subsref:ItemDataType">
<xs:attributeGroup ref="dst:PaginationResponseAttributeGroup"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<!--endsec(queryresp)-->  
<!--sec(mod)-->  
<xs:complexType name="ModifyType">
<xs:complexContent>
<xs:extension base="dst:RequestType">
<xs:sequence>
<xs:element ref="subsref:Subscription" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="subsref:ModifyItem" minOccurs="1" maxOccurs="unbounded"/>
<xs:element ref="subsref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<!--endsec(modaux)-->  
<xs:element name="ModifyItem" type="subsref:ModifyItemType"/>
<xs:complexType name="ModifyItemType">
<xs:complexContent>
<xs:extension base="dst:RequestType">
<xs:sequence>
<xs:element ref="subsref:Subscription" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="subsref:ModifyItem" minOccurs="1" maxOccurs="unbounded"/>
<xs:element ref="subsref:ResultQuery" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<!--endsec(mod)-->
<xs:sequence>
  <xs:element ref="subsref:Select" minOccurs="0" maxOccurs="1"/>
  <xs:element ref="subsref:NewData" minOccurs="0" maxOccurs="1"/>
</xs:sequence>
<xs:attributeGroup ref="dst:ModifyItemAttributeGroup"/>
<xs:complexType>
  <xs:extension base="subsref:DataResponseType"/>
</xs:complexType>
<xs:complexType name="ModifyResponseType">
  <xs:complexContent>
    <xs:extension base="dst:RequestType">
      <xs:sequence>
        <xs:element ref="subsref:Select" minOccurs="0" maxOccurs="1"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="DeleteItem" type="subsref:DeleteItemType"/>
<xs:complexType name="DeleteItemType">
  <xs:complexContent>
    <xs:extension base="dst:DeleteItemBaseType">
      <xs:sequence>
        <xs:element ref="subsref:Select" minOccurs="0" maxOccurs="1"/>
        <xs:element name="Sort" minOccurs="0" maxOccurs="1" type="subsref:SortType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="DeleteResponseType">
  <xs:complexContent>
    <xs:extension base="lu:ResponseType"/>
  </xs:complexContent>
</xs:complexType>
<xs:element name="Select" type="subsref:SelectType"/>
<xs:element name="ResultQuery" type="subsref:ResultQueryType"/>
<xs:complexType name="ResultQueryType">
  <xs:complexContent>
    <xs:extension base="dst:ResultQueryBaseType">
      <xs:sequence>
        <xs:element ref="subsref:Select" minOccurs="0" maxOccurs="1"/>
        <xs:element name="Sort" minOccurs="0" maxOccurs="1" type="subsref:SortType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="ItemData" type="subsref:ItemDataType"/>
<xs:complexType name="ItemDataType">
  <xs:complexContent>
    <xs:extension base="subsref:AppDataType">
      <xs:attributeGroup ref="dst:ItemDataAttributeGroup"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="Aggregation" minOccurs="0" maxOccurs="1" type="subsref:AggregationType"/>
<xs:element name="Trigger" minOccurs="0" maxOccurs="1" type="subsref:TriggerType"/>
<xs:element name="Notification" type="subsref:NotificationType"/>
<xs:complexType name="NotificationType">
  <xs:complexContent>
    <xs:extension base="subs:NotificationType">
      <xs:sequence>
        <xs:element ref="subsref:ItemData" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="NotifyResponseType">
  <xs:complexContent>
    <xs:extension base="subs:NotifyResponseType"/>
  </xs:complexContent>
</xs:complexType>

8.4. Subscriptions Utility Schema

The formal utility schema follows
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  targetNamespace="urn:liberty:subs:2006-02"
  xmlns:subs="urn:liberty:subs:2006-02"
  xmlns:lu="urn:liberty:util:2005-11"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
  <xs:import namespace="urn:liberty:util:2005-11"
    schemaLocation="liberty-idwsf-utility-v2.0.xsd"/>
  <!--sec(subscr)-->
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:enumeration value="Yes"/>
    <xs:enumeration value="No"/>
    <xs:enumeration value="YesWithCommonAttributes"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType>
  <xs:element name="RefItem" type="subs:RefItemType"/>
  <xs:complexType name="RefItemType">
    <xs:attribute name="subscriptionID" use="optional" type="lu:IDType"/>
    <xs:attribute ref="lu:itemIDRef" use="required"/>
  </xs:complexType>
</xs:complexType>

<xs:complexType name="NotifyAttributeGroup">
  <xs:attribute name="timeStamp" use="optional" type="xs:dateTime"/>
</xs:complexType>

<xs:complexType name="NotificationType">
  <xs:sequence>
    <xs:element ref="lu:TestResult" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="id" use="optional" type="xs:ID"/>
  <xs:attribute name="subscriptionID" use="required" type="lu:IDType"/>
  <xs:attribute name="expires" use="optional" type="xs:dateTime"/>
  <xs:attribute name="endReason" use="optional" type="xs:anyURI"/>
</xs:complexType>

<xs:complexType name="NotifyResponseType">
  <xs:complexContent>
    <xs:extension base="lu:ResponseType"/>
  </xs:complexContent>
</xs:complexType>
References

Normative


Informative