Web Services for Remote Portlets 1.0 Primer

Committee Draft 1.00, 03 December 2004

Document identifier:
wsrp-1.0-primer-1.0

Location:
http://www.oasis-open.org/committees/wsrp

Editors:
Subbu Allamaraju, *BEA Systems Inc* <Subbu.Allamaraju@bea.com>
Rex Brooks, *Starbourne Communications* <rexb@starbourne.com>

Contributors:
Atul Batra, *Sun Microsystems Inc* <Atul.Batra@sun.com>
Clinton Davidson, *Plumtree Software* <Clinton.Davidson@plumtree.com>
Alan Kropp, *Vignette Corporation* <akropp@vignette.com>
Gil Tayar, *WebCollage* <Gil.Tayar@webcollage.com>

Abstract:
This is the WSRP 1.0 primer. The purpose of this document is to provide a tutorial-oriented explanation of the main concepts of the WSRP 1.0 specification. Although this primer is not normative, its tutorial approach is intended as a guide for implementers and other technical readers. This document explains the abstractions of the specification using sample scenarios, and provides guidance and best practices for implementers and advanced users of the WSRP Specification.

Status:
This version is a committee draft of the non-normative WSRP 1.0 primer. Comments about points needing clarification are much appreciated and may be emailed to wsrp-primer@lists.oasis-open.org.

If you are on wsrp@lists.oasis-open.org list for committee members, send comments there. If you are not on that list, subscribe to wsrp-comment@lists.oasis-open.org list and send comments there. To subscribe, send an email message to wsrp-comment-request@lists.oasis-open.org with the word "subscribe" as the body of the message.

1 Introduction

Web Services for Remote Portlets (WSRP) is a web services protocol for aggregating content and interactive web applications from remote sources. Web Services for Remote Portlets 1.0 Primer is a non-normative document intended to help interpret the WSRP 1.0 Specification [1] with usage scenarios and typical interactions that must happen to achieve such aggregation. There are numerous sources of high-level introductory information about WSRP 1.0, including the introductory section of the specification itself, and the WSRP White Paper [2]. If you are reading about WSRP for the first time, we encourage you to explore these resources before proceeding with the extended examples and explanations contained in this primer.

The guiding perspective on which WSRP specification was built should be of primary interest to potential implementers. This perspective is framed by the question of what problems WSRP is intended to solve. The specification's procedural approach addresses the following main areas:

1. **Standard remote content protocol:** As a standard remote content protocol, WSRP replaces many proprietary, product-specific solutions for aggregating remote content and interactive applications. This benefits all parties, consumers (e.g. portals), portlet developers (developers of content and applications), and producers (applications hosting portlets) as well.
2. **Promote rigorous portlet implementations:** WSRP raises the bar of conformance for this standard in many respects for what constitutes a good or effective portlet implementation. The specification makes specific recommendations regarding markup fragment rules, representing state, ensuring security, etc., with an eye toward maximizing the usefulness and integrity of portlet services. This is not to suggest that WSRP mandates a one size fits all approach.
3. **Framework for sophisticated scenarios:** WSRP 1.0 is the foundation on which increasingly sophisticated implementations can be specified. These include the ability for Consumers to customize a portlet's content, and to create application process flows that coordinate the activities of multiple portlets, from multiple portlet Producers.

WSRP builds on a few fundamental standards, most notably XML, SOAP and WSDL, while allowing for the implementation of evolving standards, to deliver a protocol rich in abstractions and operations that web service implementers and portlet Consumers require.

2 Basic Scenario

The WSRP specification uses the terms Producer and Consumer to describe parties implementing the specification.

The Producer is a web service that offers one or more portlets and implements various WSRP interfaces/operations. Depending on the implementation, a producer may offer just one portlet, or may provide a runtime (or a container) for deploying and managing several portlets.

The Consumer is a web service client that invokes producer-offered WSRP web services and provides an environment for users to interact with portlets offered by one or more such Producers.

You can use WSRP to implement a very broad range of portlet Producers and Consumers. However, in this primer, for the sake of simplicity, we use simpler examples. It is not our intention to address the entire range of problems that WSRP can solve, or to replace the material already in the specification. Therefore, when you uncover your own questions, and discover that any particular question is not discussed here, we suggest that you have a copy of the specification available for quick reference.

In this Primer, we use interactions between two parties, viz., P Inc (a WSRP Producer), and C Inc (a WSRP Consumer) to discuss various WSRP interfaces.

In the examples we will use, P Inc is a financial services company, providing services online to their customers and partners. C Inc is on online portal company, providing personalized collaboration, banking, and financial services. C Inc offers these services to end-users by subscription.

P Inc would like to host a number of applications including a web based portfolio management application. C Inc would like to offer this application to its end users via its portal pages.

In order to offer this portfolio management application to end users, C Inc and P Inc agree on the following:

1. P Inc makes some metadata of the portfolio management application available to C Inc. C Inc. uses this metadata to prepare a page that users can use to manage their portfolios.
2. A user of C Inc visits C Inc's web site, and clicks on a link to portfolio management application.
3. C Inc then sends a request to P Inc to get the initial view of the portfolio management application. P Inc then responds by returning HTML markup that represents the first page of the application.
4. C Inc processes the returned markup and prepares it for aggregation. If the returned markup has links and forms, C Inc transforms the markup such that such links and forms, when activated return to C Inc.
5. C Inc aggregates the markup into a page, writes it into the response of the browser's connection, and transmits the aggregated page to the user's browser.
6. User reviews the page, and finds a form to submit a new stock symbol. User then fills in the symbol of a stock and other details, and submits the form.
7. C Inc receives the request containing the form data submitted by the user. Upon determining that this request is meant for the portfolio management application, C Inc sends another request to P Inc to process the user interaction.
8. P Inc processes the user interaction, adds the symbol to user's portfolio, and returns new state for the portfolio.
9. C Inc then sends a request to get the changed markup based on the current state of the portfolio. P Inc generates markup and returns.
10. C Inc then repeats steps (d) and (e).
11. User receives a new page containing the updated portfolio.

This scenario captures some of the essentials of the WSRP 1.0 Specification. Instead of developing a proprietary application protocol to accomplish the above steps, P Inc and C Inc can agree to use WSRP as the protocol. In this scenario, P Inc is a WSRP Producer offering portlets, and C Inc is a WSRP Consumer consuming portlets and aggregating portlets for users to access aggregated portlet pages. The portfolio management application is a Portlet offered by the Producer.

Note: This version of the Primer will not address how C Inc may discover the web service end-point offered by P Inc.

To implement this scenario, P Inc and C Inc can use WSRP to define various interactions, with P Inc implementing the following required WSRP interfaces and operations:

1. **Service Description Interface:** P Inc implements this interface to provide metadata of itself and the list of portlets it offers. C Inc invokes the `getServiceDescription` operation of this interface to obtain this metadata in step (a) of the above scenario.
2. **Markup Interface:** To generate markup and to process interaction requests, P Inc implements the markup interface specified by WSRP. C Inc invokes the `getMarkup` operation of this interface to obtain the portlet's markup in steps (c) and (i), and invokes the `performBlockingInteraction` operation to propagate user's interactions to P Inc in step (g).

By implementing these interfaces, and agreeing to conform to WSRP, both P Inc and C Inc can use a standard mechanism to offer and consume portlets. In addition, P Inc can offer the same portlet to X Inc as long as X Inc adheres to WSRP, and C Inc can consume portlets offered by Y Inc provided Y Inc also implements WSRP interfaces.

The Service Description and Markup interfaces are the two required interfaces that any WSRP Producer must implement. In addition, WSRP specifies the following optional interfaces:

1. **Registration Interface:** Registration interface provides an in-band mechanism for a Consumer to register with a Producer and let the Producer customize its behavior for each Consumer based on the registration information. As described in Section 4, WSRP also allows out-of-band mechanisms for registration, and Producers that do not require registration.
2. **Portlet Management Interface:** The Portlet Management interface allows Consumers to clone/destroy portlets, and also customize portlets by changing any associated properties.

In the following sections, we will discuss these interfaces in detail by considering each aspect of the above scenario. To keep the discussions focused on the purpose and usage of these interfaces, we postpone any discussion on faults to Section 8.1. Refer to the WSRP specification for the list of faults that various operations in each of the interfaces may return. Also, note that the data structures used in the messages through out this Primer do not necessarily include all optional elements.

2.1 Useful References

You can find the normative WSDL of all WSRP interfaces in [3], and the data types in [4]. We encourage you to refer to these documents for more complete descriptions of various messages presented in this Primer.

If you have questions about implementing WSRP, post your questions to the wsrp-dev@lists.oasis-open.org mailing list.

During the development of WSRP, the WSRP technical committee experimented with various well-known web services stacks such as .NET, JAX-RPC reference implementation, Apache Axis etc. For a discussion of stack related issues considered during the development of WSRL WSDL and data types, refer to [5].

To verify that your implementation conforms to the WSRP Specification, you can use the conformance test kit [6]. You can also review the conformance requirements at [7].

2.2 Conventions

Throughout this Primer, we use illustrative scenarios, sample SOAP message fragments, and highlight implementation choices that Producers and Consumers could make. We use the following formatting conventions to aide readability:

1. **Sample scenario:** In this Primer, we use the following format to describe sample scenarios between a Producer and a Consumer.

   Scenario: This is a sample scenario, and describes a particular interaction between P Inc and C Inc.

2. **Messages:** We use the following format to present sample SOAP message fragments between a Producer and a Consumer. Note that these message fragments are illustrative

and are not valid SOAP messages.

```
<urn:getServiceDescription
  xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext xsi:nil="true"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
</urn:getServiceDescription>
```

3. **Hints:** We use the following format to highlight possible implementation choices and/or hints for advanced readers.

   This is a hint.

4. **XML elements and attributes:** We use `fixed-width` font to indicate XML elements and attributes used in running text.

3 Service Description Interface

3.1 Introduction

In order to set up a Consumer to aggregate portlets offered by a Producer, the Consumer must first obtain a description of the Producer, and the list of portlets that the Producer offers. Based this information, the Consumer will be able determine if it can successfully aggregate those portlets, and setup its environment (for example, a page aggregating portlets) for such aggregation.

The `getServiceDescription` operation of the service description interface provides the Producer's metadata including the list of offered portlets and their properties.

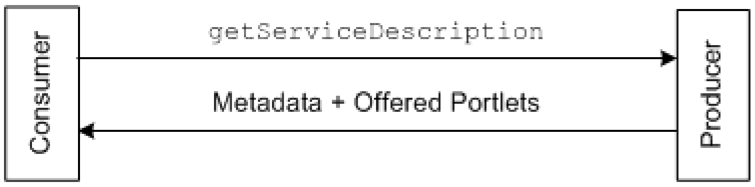

Figure 1: Getting Service Description

All portlet Producers are required to implement this operation. Usually, the first request a Consumer ever sends to a Producer is the `getServiceDescription` request. This operation returns the following:

1. Producer's Capabilities (i.e., metadata): The response of this operation indicates to the Consumer whether the Producer requires registration to access its portlets, whether the Consumer must explicitly initialize cookies for all markup related operations, the locales supported by the producer etc. We shall discuss the details of this metadata as we introduce various other WSRP interfaces. This information helps you set up a Consumer to interact with the Producer.
2. Offered Portlets: The response of this operation also includes a list of portlets that the Producer offers. The portlet metadata includes a unique handle, modes, window states, and content types supported by the portlet, in addition to a description of the portlet. In this sense, the Producer acts as a portlet repository that the Consumer must access to discover portlets.

3.2 Descriptions of Operations

Consider the following scenario.

C Inc would like to discover the capabilities of P Inc, and the list of portlets offered by P Inc.

Scenario 1: Discover Portlets

In order to get P Inc's service description, C Inc must first send a `getServiceDescription` request to P Inc. Here is the most basic form of a `getServiceDescription` request that a consumer could send to a producer.

```
<urn:getServiceDescription
  xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext xsi:nil="true"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
</urn:getServiceDescription>
```

Message 1: Service Description Request

To this request, P Inc responds with a `getServiceDescriptionResponse` document that includes the following:

1. Registration is not required: P Inc does not require C Inc to register before sending other WSRP requests. We shall discuss registration in Section 4.
2. Supported Locales: P Inc supports `en` and `en-US` locales. This does not necessarily mean that portlets offered by this Producer are limited to generating markup in these locales, but simply that the Producer's metadata is available in these locales.
3. Offered Portlets: P Inc offers a single portlet, uniquely identified by a `portletHandle` "portfolioManager". The `portletHandle` is an opaque reference assigned by the Producer, and both the Consumer and the Producer use this handle to refer to this portlet in all interactions
4. Supported MIME types: The portfolio manager portlet returns content of MIME type `text/html`. As with locales, this portlet is not limited to generating markup in this MIME type.
5. Modes and window states supported: Modes and window states indicate the many ways a portlet may be rendered. The portfolio manager portlet could be rendered in the `wsrp:view` mode, and `wsrp:normal`, `wsrp:minimized`, and `wsrp:maximized` window states.
6. Description: The description of the portfolio manager portlet also includes a description and a title. The Consumer may use these values for describing or presenting this portlet to its end users. For example, the Consumer may provide a title bar that includes this title with the portlet's markup.

Here is the response message from P Inc. containing the above data:

```
<urn:getServiceDescriptionResponse
  xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:requiresRegistration>false</urn:requiresRegistration>
  <urn:offeredPortlets>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
    <urn:markupTypes>
      <urn:mimeType>text/html</urn:mimeType>
      <urn:modes>wsrp:view</urn:modes>
      <urn:windowStates>wsrp:normal</urn:windowStates>
      <urn:windowStates>wsrp:minimized</urn:windowStates>
      <urn:windowStates>wsrp:maximized</urn:windowStates>
      <urn:locales>en</urn:locales>
      <urn:locales>en-US</urn:locales>
    </urn:markupTypes>
    <urn:description xml:lang="en">
      <urn:value>Manages portfolios</urn:value>
    </urn:description>
    <urn:title xml:lang="en">
      <urn:value>Manage Your Portfolios</urn:value>
    </urn:title>
  </urn:offeredPortlets>
  <urn:locales>en</urn:locales>
  <urn:locales>en-US</urn:locales>
</urn:getServiceDescriptionResponse
```

Message 2: Service Description Response

C Inc can now use this response to setup a page to aggregate this portlet, and offer that page to its users.

The portlets described in the `getServiceDescriptionResponse` are &qout;producer-offered" portlets. In Section 5.2.3 and 6.2.4, we will see how Consumers can cause cloning of these portlets to create consumer-configured portlets. This distinction is important because all Consumers are allowed to access/use producer-offered portlets and therefore no Consumer is allowed to customize them.

3.3 Recommendations

Producers' capabilities as well as the metadata of portlets offered by a Producer may change over time. When such changes happen, it is very likely that Consumers/portlets may not function correctly. To prevent such failures, and since WSRP does not provide a mechanism for Producers to notify Consumers of such changes automatically, we recommend that Producers keep such metadata unchanged, or notify Consumers of changes. In general, the guiding principle is that Producers must treat service descriptions and portlet descriptions as a published contract with all the Consumers.

Secondly, a Producer offering a fixed set of portlets with the same behavior to all Consumers without requiring any knowledge of the Consumer may follow the style of service description response described in this section. However, in most cases, Producers and Consumers may find it necessary to customize their behavior based on certain properties of the other party. For example, P Inc may want to offer the portfolio manager portlet only to those Consumers that enter into a service contract. Another Producer may want to customize its responses depending on the capabilities of the Consumer. As discussed in Section 4, Consumers and Producers may use the notion of registration to deal with such scenarios.

4 Registration Interface

4.1 Introduction

The purpose of the registration interface is to provide a means within the WSRP protocol for a Consumer to register with a Producer. Registration allows the Producer to associate portlets and any portlet customization data with the Consumer that is interacting with it. The Producer can also use the registration context to scope the artifacts offered/created during interactions to Consumers that caused those interactions. Note that the purpose of registration is not to uniquely *identify* a Consumer, but to establish a scope for a Consumer's use of a Producer.

The WSRP specification does not specify/restrict any possible application of registration. Here are some possible applications:

1. Consumer can use registration to let the Producer know of its capabilities. The Producer can use such metadata to tailor its portlets.
2. Producer can keep track of portlets used by each Consumer, by associating any persistent state of portlets and any cloned portlets with the Consumer's registration.
3. Producer can tailor the list of portlets offered to each Consumer. For example, Producers may offer a separate set of portlets for each Consumer.
4. Producer may offer/deny certain capabilities (such as the ability to customize or clone portlets) for a given Consumer.

Depending on the nature of services offered by the Producer and the nature of the business, the registration process may be as simple as sending a registration request to a Producer using the Registration interface, or may be as complex as fulfilling legal, billing and other contractual obligations to establish registration. Keeping this in mind, the WSRP 1.0 specification considers two forms of registration:

1. In-band registration: In this process, using the Registration interface, the Consumer sends a request to register to the Producer, along with any properties required by the Producer. If required by the Producer, the Consumer may have to go through some out-of-band communications before being able to send such a request.
2. Out-of-band registration: The Producer and Consumer go through Producer/Consumer specific business processes (such as other business web applications, or email or other manual means) to establish registration. Since the actual processes tend to be very specific to each Consumer and Producer, WSRP specification does not attempt to standardize these processes. Further discussion on out-of-band registration is therefore outside the scope of this Primer.

By following one of these forms of registration, the Consumer obtains a `registrationContext` from the Producer. The `registrationContext` includes a `registrationHandle`, and optionally some `registrationState`. Of these, the `registrationHandle` is a unique handle assigned by the Producer to the Consumer and remains unchanged during the lifetime of a Consumer's registration with a Producer. The optional `registrationState` contains any persistent state for the registration which the Producer requires the Consumer to store and resupply on future invocations.

The registration interface specifies the following operations:

1. `register`: This operation lets a Consumer register with a Producer. The Consumer may have to supply certain producer-specified registration properties for registration. Upon registration, the Producer assigns a unique `registrationContext` for the Consumer. The Consumer must then supply this `registrationContext` with each request it makes to the Producer.
2. `modifyRegistration`: This operation lets a Consumer modify an existing registration. If required, the Consumer may supply registration properties with this request. The Producer may not assign a new `registrationHandle` to the Consumer, but may return new `registrationState`.
3. `deregister`: This operation lets a Consumer terminate a registration.

A Producer indicates that registration is required and that certain data is required for registration via its response to a `getServiceDescription` request.

4.2 Descriptions of Operations

4.2.1 Registration

Let us now consider the following variation to Scenario 1 to see some of the details of registration.

P Inc requires that only registered Consumers can see its metadata or use any of its offered portlets. P Inc also requires Consumers to fulfill a service agreement and certain contractual obligations prior to registration.

C Inc enters into a service agreement with P Inc and fulfills all the necessary obligations. P Inc then issues a service ID to C Inc. P Inc. requires that C Inc supply C Inc's DUNS (Data Universal Numbering System, assigned by Dun and Bradstreet Corp for enterprises) number and the service ID for registration. Note that the transactions establishing these numbers happen outside the scope of WSRP.

C Inc then uses the in-band registration procedure to register with P Inc by supplying its DUNS number and the service ID.

Scenario 2: Registration

Since P Inc now requires registration, P Inc sends the following response to the previous simple `getServiceDescription` request from C Inc.

```
<urn:getServiceDescriptionResponse xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:requiresRegistration>true</urn:requiresRegistration>
  <urn:registrationPropertyDescription>
  <urn:propertyDescriptions type="urn:stringValue" name="dunsNum">
    <urn:label xml:lang="en">
      <urn:value>DUNS Number</urn:value>
    </urn:label>
  </urn:propertyDescriptions>
  <urn:propertyDescriptions type="urn:stringValue" name="serviceId">
    <urn:label xml:lang="en">
      <urn:value>Service ID</urn:value>
    </urn:label>
  </urn:propertyDescriptions>
  </urn:registrationPropertyDescription>
</urn:getServiceDescriptionResponse>
```

Message 3: Service Description Response from Producer Requiring Registration

The first point to note from this response is that P Inc now requires registration, and that it requires Consumers to supply two registration properties for registration.

In this response, both the properties are of type `stringValue` in the `urn:oasis:names:tc:wsrp:v1:types` namespace. Producers can use either the `stringValue` type or any arbitrary schema type (other than those defined in the `urn:oasis:names:tc:wsrp:v1:types` namespace) to describe a registration property.

Other than these requirements, this service description response does not provide much information about P Inc. Specifically P Inc chose to omit portlet metadata in its response.

C Inc then sends the following request to P Inc supplying the DUNS number and service ID that P Inc. assigned.

```
<urn:register xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:consumerName>C Inc</urn:consumerName>
  <urn:consumerAgent>C Inc Portal.1.0</urn:consumerAgent>
  <urn:methodGetSupported>false</urn:methodGetSupported>
  <urn:registrationProperties xml:lang="en" name="dunsNum">
    <urn:stringValue>CIncDunsNumber</urn:stringValue>
  </urn:registrationProperties>
  <urn:registrationProperties xml:lang="en" name="serviceId">
    <urn:stringValue>CIncServiceID</urn:stringValue>
  </urn:registrationProperties>
</urn:register>
```

Message 4: Registration Request

In addition to the registration properties, the Consumer sends some additional metadata of itself:

1. Consumer name: Name of the Consumer
2. Consumer agent: Name and version of the Consumer
3. Method GET support: A boolean to indicate if the consumer can aggregate markup containing forms with method GET.

Support for forms with method GET requires some special considerations. This is because Consumer implementations are likely to embed implementation-specific parameters in URLs.

However, as any web developer would be familiar with, HTML forms using GET as the method type cannot include query parameters in action URLs. To support such forms, Consumers will have to use hidden parameters or cookies, or some other technique to embed implementation-specific data. If a given Consumer is not capable of supporting such forms, it can indicate so to the Producer by supplying false for the `methodGetSupported` element. The Producer can then avoid offering portlets that generate forms with method GET to the Consumer.

Refer to the WSRP specification for a list of other optional data that the Consumer can supply during registration.

The Producer P Inc validates this request for registration, creates a registration context, and returns the following response:

```
<urn:registerResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
</urn:registerResponse>
```

Message 5: Registration Response

In this response, the `registrationContext` represents a registration relationship between the P Inc and C Inc, and contains a `registrationHandle` assigned to C Inc by P Inc. Once a Consumer obtains a `registrationContext`, the Consumer must supply the `registrationContext` with all subsequent requests to the Producer. For this reason, the Consumer must persistently maintain the `registrationContext` for all future invocations.

In response to a registration request, the Producer stores the Consumer's data persistently, creates a `registrationHandle` and returns this handle via a `RegistrationContext`.

If the Producer is not capable of managing persistent storage of registration data, the Producer can choose to return all such data to be persisted as `registrationState` to the Consumer. The Consumer would then be responsible for storing the `registrationState` along with the `registrationHandle` and return these to the Producer in future invocations.

Having completed the registration process, C Inc can now send another service description request to P Inc, this time supplying the `registrationContext`.

```
<urn:getServiceDescription xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
</urn:getServiceDescription>
```

Message 6: Service Description Request after Registration

The `registrationContext` in this request helps P Inc to tailor its response. Upon validating the `registrationContext`, P Inc sends a response that includes portlets it offers.

```
<urn:getServiceDescriptionResponse xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:requiresRegistration>true</urn:requiresRegistration>
  <urn:offeredPortlets>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
    <urn:markupTypes> <urn:mimeType>text/html</urn:mimeType>
      <urn:modes>wsrp:view</urn:modes>
      <urn:windowStates>wsrp:normal</urn:windowStates>
      <urn:windowStates>wsrp:minimized</urn:windowStates>
      <urn:windowStates>wsrp:maximized</urn:windowStates>
      <urn:locales>en</urn:locales> <urn:locales>en-US</urn:locales>
    </urn:markupTypes>
    <urn:description xml:lang="en">
      <urn:value>Manages portfolios</urn:value>
    </urn:description>
    <urn:title xml:lang="en">
      <urn:value>Manage Your Portfolios</urn:value>
    </urn:title>
  </urn:offeredPortlets>
  <urn:requiresInitCookie>none</urn:requiresInitCookie>
  <urn:registrationPropertyDescription>
    <urn:propertyDescriptions type="xs:string" name="dunsNum">
      <urn:label xml:lang="en">
        <urn:value>DUNS Number</urn:value>
      </urn:label>
    </urn:propertyDescriptions>
    <urn:propertyDescriptions type="xs:string" name="serviceId">
      <urn:label xml:lang="en">
        <urn:value>Service ID</urn:value>
      </urn:label>
    </urn:propertyDescriptions>
  </urn:registrationPropertyDescription>
  <urn:locales>en</urn:locales>
  <urn:locales>en-US</urn:locales>
</urn:getServiceDescriptionResponse>
```

Message 7: Service Description Response after Registration

Note that this message is similar to the response shown in Message 2, but has only been received after C Inc fulfilled P Inc's registration requirements.

4.2.2 Modifying a Registration

Once a Consumer establishes a registration relationship with a Producer, in some conditions, Consumers may have to modify an existing registration relationship either to be able to continue to use a Producer or alter certain properties of the registration. Here are some possible situations that may warrant such an operation:

1. Changes to registration properties: Producer requires a different set of registration properties than were previously used at the time of registration. In this case, the Consumer may be required to supply the new registration properties to be able to continue to use the Producer.
2. Changes to Consumer's capabilities: Consumer has decided to send optional data not previously supplied.

Consider the following scenario.

P Inc now requires an email address of the Consumer to be supplied for each Consumer's registration.

C Inc sends a `modifyRegistration` request to P Inc supplying the email address.

Scenario 3: Modify Registration

To implement this scenario, P Inc has the following options:

1. P Inc may return `OperationFailed` faults until C Inc. supplies the new property (email address).
2. Inform C Inc (out of band) that a new property is required.

In the former case, C Inc will have to send `getServiceDescription` request (without the `registrationContext`) to discover the current set of registration properties. Upon discovering the new registration property, C Inc sends the following `modifyRegistration` request to P Inc.

```
<urn:modifyRegistration xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:registrationData>
    <urn:consumerName>C Inc</urn:consumerName>
    <urn:consumerAgent>C Inc Portal.1.0</urn:consumerAgent>
    <urn:methodGetSupported>false</urn:methodGetSupported>
    <urn:registrationProperties xml:lang="en" name="dunsNum">
      <urn:stringValue>CIncDuncNumber</urn:stringValue>
    </urn:registrationProperties>
    <urn:registrationProperties xml:lang="en" name="serviceId">
      <urn:stringValue>CIncServiceID</urn:stringValue>
    </urn:registrationProperties>
    <urn:registrationProperties xml:lang="en" name="email">
      <urn:stringValue>admin@cinc.com</urn:stringValue>
    </urn:registrationProperties>
  </urn:registrationData>
</urn:modifyRegistration>
```

Message 8: Request to Modify Registration with an Additional Registration Property

This request is similar to the registration request in Message 4 except for a new value for a registration property.

P Inc validates the new values for registration properties, modifies registration, and responds with the following message:

```
<urn:modifyRegistrationResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types"/>
```

Message 9: Modify Registration Response

The response simply indicates that P Inc. has accepted the request for modification of registration.

If the Producer is not capable of managing persistent storage of the registration data, the Producer may return the updated registration state as `registrationState` in the `modifyRegistrationResponse`.

After modifying the registration relationship, C Inc can continue to use the `registrationHandle`.

4.2.3 Terminating a Registration

Registration relationships are not permanent. Either the Consumer or the Producer may terminate a registration relationship. Here are some scenarios that could prompt for a termination:

1. Consumer decides not to aggregate portlets from a Producer. To notify the Producer that it no longer has to manage any cloned portlets and other persistent state created during the course of the registration, the Consumer can send a request to the Producer to terminate the registration
2. Producer decides not to offer portlets to a given Consumer. Producer may unilaterally terminate the registration, or notify (out of band) the Consumer that it will no longer offer any portlets to the Consumer.
3. Consumer wants to recreate registration due to some changes in Consumer's environment.

In cases (a) and (c) above, the Consumer can send a `deregister` request to the Producer to terminate the current registration relationship. Note that, Consumer is not required to `deregister` always. However, by formally deregistering, Consumer lets the Producer cleanup any persistent data maintained for that registration.

Consider that C Inc now wants to terminate its registration with P Inc. In order to do so, C Inc sends a `deregister` request with its `registrationHandle` to P Inc as shown below.

```
<urn:deregister xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
</urn:deregister>
```

Message 10: Deregister Request

P Inc then deregisters C Inc and cleans up any resources/state created (including any cloned portlets) for C Inc, and returns the following response.

```
<urn:deregisterResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types"/>
```

Message 11: Deregister Response

After this step, C Inc can no longer use the `registrationHandle` "CIncRegnHandle" for its requests to P Inc.

Terminating a registration has some consequences for both the Producer and the Consumer:

1. Consumer can delete any registration-specific persistent state created.
2. Producer can also delete any registration-specific persistent state created.
3. Consumer can register again with the Producer, but may not refer to any cloned portlets created during a terminated registration.

Figure 2 shows possible interactions between C Inc and P Inc and the registration life cycle.

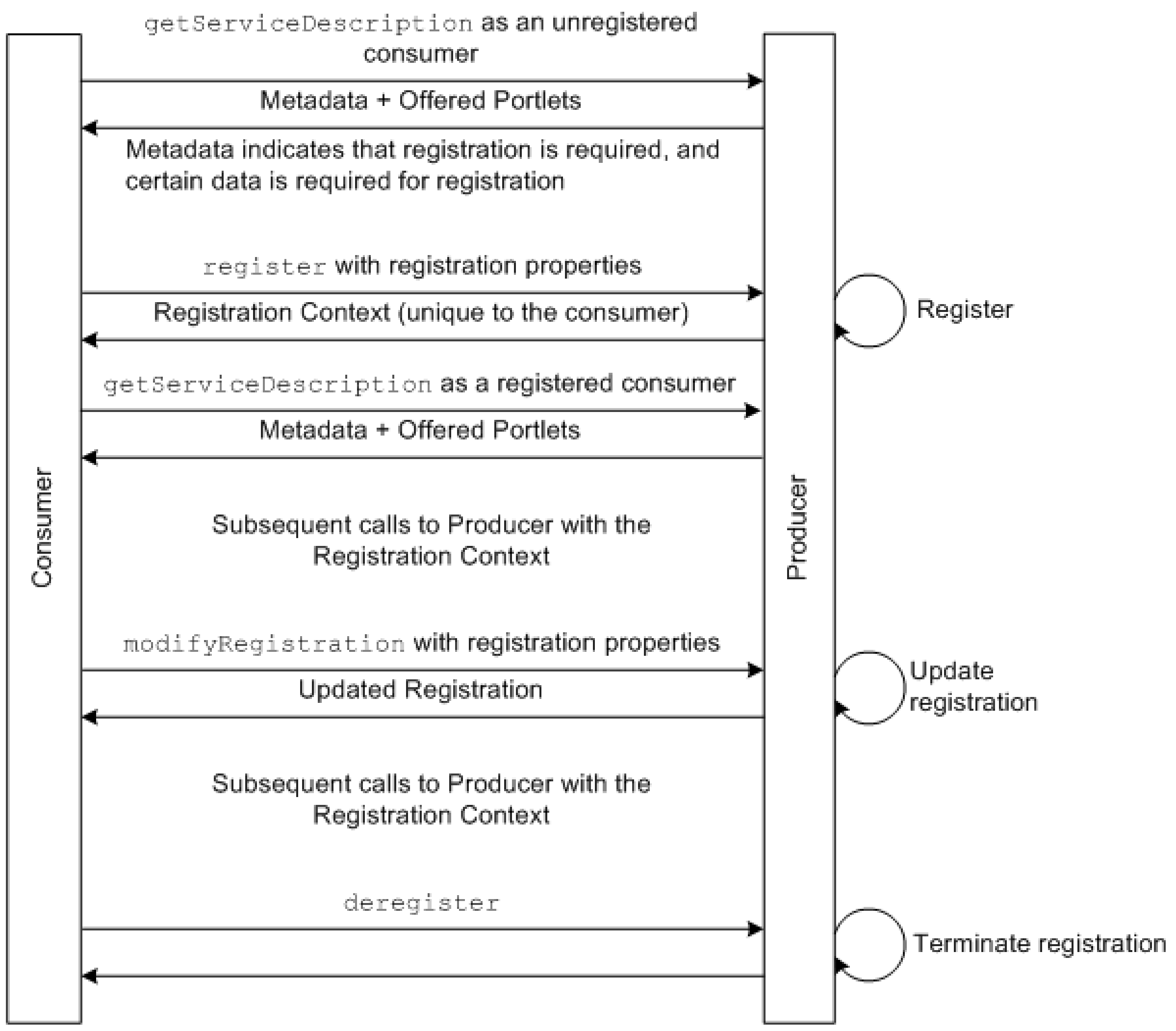
getServiceDescription as an unregistered consumer
Metadata + Offered Portlets
Metadata indicates that registration is required, and certain data is required for registration
register with registration properties
Registration Context (unique to the consumer)
getServiceDescription as a registered consumer
Metadata + Offered Portlets
Subsequent calls to Producer with the Registration Context
modifyRegistration with registration properties
Updated Registration
Subsequent calls to Producer with the Registration Context
deregister
Consumer
Producer
Register
Update registration
Terminate registration

Figure 2: Registration Impact on Consumer-Producer Interactions

4.3 Recommendations

Once you establish a registration between a Producer and a Consumer, the registration becomes long lasting. This is because the registration relationship is tied to any persistent data stored on both the Producer and Consumer. Deregistering a Consumer from a Producer will invalidate such persistent data, and your users will not be able to use their portlet settings.

Due to the reasons discussed in Section 4.2.2, if a Consumer modifies an existing registration, we recommend that the Consumer obtain the service description from the Producer again. This is because Producers may use registration data to tailor service description to each Consumer.

Note that registration does not address the question of Consumer identity in the security sense. You may need other means of trust to address the question of Consumer identity.

5 Markup Interface

5.1 Introduction

In order for a Consumer to generate a page that aggregates portlets offered by one or more Producers, the Consumer must first obtain the markup of each portlet from each Producer. As users interact with the portlet (e.g. by submitting a form from the portlet's markup), the Consumer must be able to send such interaction requests to the Producer, and then receive markup reflecting the user interaction. The Markup interface specifies operations for achieving these tasks. All Producers are required to support this interface.

The markup interface includes the following operations:

1. `getMarkup`: The purpose of this operation is let the Consumer collect markup fragment for for a given portlet.
2. `performBlockingInteraction`: Consumers use this operation to dispatch each user interaction to the Producer hosting the portlet that sourced the markup the user is interacting with.
3. `initCookie`: The purpose of this operation is to let the Producer initialize any cookies, and set those cookies to the Consumer. As we shall discuss shortly, using this operation Producers may return cookies to Consumers for load-balancing or other implementation specific purposes.
4. `releaseSessions`: As we shall see shortly, during a `getMarkup` or a `performBlockingInteraction`, portlets can initialize sessions. When a portlet creates a session, the Producer returns a `sessionID` for the session to the Consumer in its response to the `getMarkup` or `performBlockingInteraction` response. Consumers can use the `releaseSessions` operation to let the Producer release those sessions.

Collecting markup and aggregating it into a page poses several challenges such as creating correct links in the markup, managing transient state, propagating user interactions etc. In this section, before reviewing the operations in this interface, let us discuss some of the most important issues that the markup interface addresses.

5.1.1 Markup Fragments

One of the primary goals of WSRP 1.0 was to allow the aggregation of multiple content units, portlets, from different sources on the same web page. HTML 4.01 and XHTML 1.1 treat pages as separate documents and disallow multiple BODY Elements, represented as <body> tags, in a single document. Therefore, the Consumer cannot aggregate individual documents with individual BODY elements from individual portlets into a single page. Portlets are therefore required to generate markup fragments.

The Consumer is most likely to aggregate the portlet’s markup fragment (such as HTML or XHTML) into a page that also includes markup from other portlets. For rules on what differentiates full markup from markup fragments, refer to Section 10.5 of the WSRP 1.0 specification. Along with the markup fragment, the Producer also returns certain properties of the markup fragment, such as the content type, character encoding, locale etc.

5.1.2 Two Step Protocol

With WSRP, the Consumer aggregates markup from different portlets hosted on Producers, and users interact with those portlets via the Consumer. The Consumer is therefore responsible for presenting portlets’ markup as well as receiving user interactions from portlets. Moreover, a user’s interaction with one portlet may affect the markup for another portlet due to any shared state between portlets.

When a user interacts with a web page, you can process the input and generate new markup (e.g., by forwarding to a new page) in a single step. However, such a single step process is not adequate for processing interactions with a page aggregating multiple portlets. During such an interaction, it is most likely that only one portlet processes the request, while all the portlets (including the one that processed the request) on the page regenerate their markup taking into account any state shared between those portlets. In an aggregated page, each portlet should be able to generate markup without any user interaction. This will allow the user to interact with one portlet, and yet see markup of all portlets on the aggregated page.

When a user interacts with a portlet's markup (e.g. by submitting a form), the Consumer forwards the interaction to the Producer identifying the portlet that generated the markup. This allows the portlet to process the user interaction (e.g. by performing queries/updates in some backend system, or updating some persistent state). During this process, the portlet processing the interaction can also update some shared state that other portlets rely on. Once this process is complete, the Consumer can get markup for each portlet from each Producer, and regenerate the aggregated page. In order to approach this sequence, WSRP specifies a two-step protocol.

The first step of the protocol is to process user interactions, as specified by the `performBlockingInteraction` operation of the WSRP protocol. The purpose of this operation is to let a portlet process the user interaction, update any transient and persistent state, and possibly return the new state to the Consumer.

The second step of the protocol is to get markup from the Producer, as specified by the `getMarkup` operation of the WSRP protocol. This operation returns the portlet's markup (a fragment), and properties of the markup (such as a preferred title, character encoding, locale and content type of the markup).

The Consumer can get markup for each portlet from each Producer even in the absence of a user interaction. This can happen, e.g. when a user visits the aggregated page for the first time, or when the user refreshes the page. The two-step protocol allows for such interaction-free rendering and using this protocol, Producers can generate markup for portlets with their current state even in the absence of any interaction for any given portlet.

As the name implies, the `performBlockingInteraction` operation is a blocking operation. That is, the Consumer waits for this operation to complete before sending `getMarkup` requests. Due to the blocking nature of this operation, Producers can allow changes made to a portlet's state during a blocking interaction visible to other portlets during their markup generation. For example, the portlet processing the blocking interaction can store some data in a relational database, and other portlets can read that data during markup generation.

5.1.3 State Management

During various `getMarkup` and `performBlockingInteraction` requests, Producers and/ or portlets can return state to Consumers. Consumers are required to supply such state in future interactions for the portlet to the Producer. The markup interface accounts for the following kinds of state:

1. Transient state: Transient state usually represents application-level state. Transient state includes the following.
    1. Navigational state: Producers can return navigational state to Consumers in response to `performBlockingInteraction` requests. Portlets can also embed navigational state in each URL in the markup. In either case, Consumer returns this navigational state to the Producer with subsequent `getMarkup` requests. Navigational state typically encapsulates data required by Producers to generate markup for a given portlet several times without having to keep track of the interaction that caused the current state of the portlet.
    2. Session State: This state corresponds to sessions initiated by portlets. When a portlet initializes a session, the Producer assigns a `sessionID` for the session, and returns it to the Consumer. Consumers return this `sessionID` in future

requests to that portlet. This mechanism is similar to HTTP state management as specified in RFC 2965 [8].

The difference between navigational state and session state is that navigational state allows the Producer to free itself from holding transient state locally and makes that state of the portlet bookmarkable by the user.

2. Persistent State: While processing interactions, Producers and/or portlets can make changes to the persistent state of the portlet. The persistent state could be properties exposed by the Producer via the portlet management interface, or some opaque state.

5.1.4 URLs in Markup

With WSRP, the Consumer is the intermediary between the Producer and the end user. Here are some advantages of the Consumer being the intermediary:

1. By being an intermediary, the Consumer can offer value-added features such as personalization, customization, security etc besides aggregation. In order to offer these features, Consumers may want to force the user to always access portlets and other Producer hosted resources via the Consumer.
2. In most cases, end users may not be able to access Producer hosted resources directly. The Consumer should therefore be able to act as a proxy for such Producer hosted resources.
3. Producers may not offer a web interface (i.e., offer http/https ports) for content.

URLs in the markup presented to the end user should therefore refer to the Consumer and not to the Producer. Either the Producer or the Consumer must take the responsibility of creating or converting URLs in the markup to refer to the Consumer. The markup interface accounts for such URL generation.

WSRP specifies two kinds of URL generation, viz. Consumer URL rewriting and Producer URL writing. Of these, the former approach requires the Consumer to rewrite URLs in the markup to refer to the Consumer. On the other hand, the latter approach lets the Producer generate URLs using Consumer supplied URL templates. We shall discuss these techniques briefly in Section 5.2.1.

5.2 Descriptions of Operations

Although the `performBlockingInteraction` operation is the first step in the two-step protocol, let us first discuss the `getMarkup` operation as it lets us probe the issues in a more natural order.

5.2.1 Get Markup

The purpose of the `getMarkup` operation is to obtain a portlet's markup. In order to discuss the semantics of this operation, consider the following scenario.

A user visits a page containing the portfolio manager portlet on C Inc's website.

C Inc sends a request to P Inc to get markup for the portfolio manager portlet.

P Inc generates a markup fragment containing a form to enter a stock symbol. This markup fragment contains some instructions to C Inc on how to transform the URLs so that the URLs refer to C Inc.

C Inc rewrites the URLs in the markup returned from P Inc, aggregates the markup into a page, and returns the page to the user.

Scenario 4: Get Initial Markup

In addition to the information about the Consumer and the portlet, the Producer needs some more data about the user (for personalizing the markup), user's device (e.g. browser), type of connection (e.g. secure or normal), and what kind of markup is acceptable to the Consumer and to the end user. The `getMarkup` request encapsulates all such information.

For the current scenario, C Inc. sends a `getMarkup` request the producer, P Inc. with the following data:

1. `RegistrationContext`: This element carries the `registrationHandle` assigned by P Inc during registration.
2. `PortletContext`: This element includes the `portletHandle`, which the Consumer uses to identify the portlet to the Producer.
3. `RuntimeContext`: This element includes the authentication mechanism (represented by userAuthetication) C Inc used to authenticate the user along with optional elements such as the `sessionID`, URL templates, etc.
4. `UserContext`: This element carries a `userContextKey` the Consumer assigns to the user. In the current scenario, C Inc sends a `nil` `UserContext` as C Inc has not assigned any `userContextKey` to the user.
5. `MarkupParams`: This element carries information about the request from the user to C Inc, such as whether the user used a secure communication channel (represented as boolean value for `secureClientCommunication`), an array for accepted locales for the markup (represented as `locale` elements), an array of accepted MIME types (represented as `mimeTypes` elements), the mode and window state for the portlet markup, accepted character sets for the markup (represented with `markupCharacterSets` elements).

Here is the request from C Inc to P Inc.

```
<urn:getMarkup xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:portletContext>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
  </urn:portletContext>
  <urn:runtimeContext>
    <urn:userAuthentication>wsrp:password</urn:userAuthentication>
  </urn:runtimeContext>
  <urn:userContext xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
  <urn:markupParams>
    <urn:secureClientCommunication>false</urn:secureClientCommunication>
    <urn:locales>en-US</urn:locales>
    <urn:mimeTypes>text/html</urn:mimeTypes>
    <urn:mode>wsrp:view</urn:mode>
    <urn:windowState>wsrp:normal</urn:windowState>
    <urn:clientData>
      <urn:userAgent>Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.6)
Gecko/20040206 Firefox/1.0</urn:userAgent>
    </urn:clientData>
    <urn:markupCharacterSets>UTF-8</urn:markupCharacterSets>
  </urn:markupParams>
</urn:getMarkup>
```

Message 12: Get Markup Request

Note the following from this request:

1. `RegistrationContext`: The `registrationHandle` is CIncRegnHandle assigned by P Inc during registration.
2. `PortletContext`: The `portletHandle` is portfolioManager.
3. `RuntimeContext`: The `userAuthentication` is `wsrp:password`. This value indicates the user supplied a username and password to authenticate with C Inc.
4. `UserContext`: The userContext is `nil`. When supplied, the `UserContext` includes a `userContextKey`. This key is an arbitrary reference assigned by C Inc for the user. Producers typically use this key for personalizing the portlets markup and behavior. In addition to this key, C Inc can also include a profile of the user (such as the name, gender, home/work information etc) and an array of userCategories with the `UserContext`. The purpose of these items is to allow the Producer to personalize the behavior and/or markup of the portlet. Refer to Section 6.17 and 6.10 of the WSRP 1.0 Specification for more details.
5. `MarkupParams`: The Boolean for `secureClientCommunication` is false indicating that the user did not user a secure connection (such as SSL) to C Inc. This element also specifies that the accepted locale is en, the accepted MIME type for markup is “text/html”, the accepted character set for the markup is UTF-8, the mode is `wsrp:view`, the windowState is `wsrp:normal` (modes and window states to be discussed later in this section). This element also includes the `clientData` element with a value for `userAgent` identifying the browser and operating system of the user.

This is a basic form of a `getMarkup` request that a Consumer could send to a Producer. To this request, P Inc responds with a `getMarkupResponse` that includes:

1. `MarkupContext`, which contains the markup for the portlet, a title, locale and MIME type of the markup.
2. `SessionContext` (optional) with a `sessionID`, and an expiry time interval (in seconds) for the `sessionID`. The Producer returns the `SessionContext` element when it establishes new session. The Consumer should supply this `sessionID` on future invocations in order to not lose state the Producer is storing for the user’s interactions. If a Consumer does not invoke this portlet before this interval, the Producer may terminate the session associated with the `sessionID`.

Here is the response from P Inc.

```
<urn:getMarkupResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:markupContext>
    <urn:mimeType>text/html; charset=UTF-8</urn:mimeType>
    <urn:markupString><![CDATA[
      <form method="post"
        action="wsrp_rewrite?wsrp-urlType=blockingAction/wsrp_rewrite"
        id="wsrp_rewrite_stockForm">
        <table border="0" width="100%">
          <tr>
            <td>Enter Stock Symbol</td>
            <td><input name="symbol"></td>
          </tr>
          <tr>
            <td><input type="submit" value="Submit"></td>
          </tr>
        </table>
      </form>
    ]]></urn:markupString>
    <urn:locale>en-US</urn:locale>
    <urn:requiresUrlRewriting>true</urn:requiresUrlRewriting>
    <urn:preferredTitle>Portfolio Manager</urn:preferredTitle>
  </urn:markupContext>
  <urn:sessionContext>
    <urn:sessionID>sessionID_1</urn:sessionID>
    <urn:expires>300</urn:expires>
  </urn:sessionContext>
</urn:getMarkupResponse>
```

Message 13: Get Markup Response

Note in our example:

1. P Inc returned a `markupContext` with a markup fragment, a preferred title "Portfolio Manager", locale "en-US" and MIME type "text/html; charset=UTF-8". C Inc can use the preferred title to render a title bar with the portlet's markup.
2. During this interaction, P Inc initialized a session with an ID "sessionID_1". C Inc is required to keep track of this ID and return it with future `getMarkup` and `performBlockingInteraction` requests for this portlet.
3. The portfolio manager portlet can process only one kind of user interaction, which is to collect a stock symbol and show the stock value.

In this response, P Inc returned the markup as a `markupString`. This is an XML-escaped string with XML entities such as `<` individually escaped as `&lt` or escaped in bulk by the use of a CDATA block as per this example.

Producers can also return the markup as Base64-encoded binary data via a `markupBinary` element in the `markupContext`.

1. Also, note the action and the ID attributes of the form in the markup. By inserting special tokens in the values of these attributes, P Inc is indicating that C Inc must rewrite the values before sending the markup to the user. Let us review this form more closely.
2. Form action URL: P Inc prefixed the value of this URL with a `wsrp_rewrite` token and suffixed with a `/wsrp_rewrite` token. The URL itself between these tokens is not a complete URL. When C Inc encounters these tokens, C Inc rewrites the URL to a string that refers to C Inc. The tokens contained between these markers are instructions on what the Consumer is to do to invoke the Producer when an end-user activates the resulting URL.
3. URL parameter `wsrp-urlType`: This is the first parameter in the URL and indicates the type of the URL. WSRP specifies three kinds of URL types – `blockingAction`, `render`, and `resource`. When the value of this parameter is `blockingAction`, C Inc converts the URL into a URL that when activated, causes a user interaction (i.e., a `performBlockingInteraction` request). When the URL type is `render`, C Inc simply needs the URL to request that it invoke `getMarkup` without an additional `performBlockingInteraction` invocation. The URL type `resource` is used for generating links to resources such as images, files etc. Refer to Section 10.2.1 of the WSRP 1.0 Specification for more details.
4. ID Parameter: P Inc prefixed the value of the ID parameter with the `wsrp_rewrite_` token. This is an indication to C Inc that it must rewrite the value `stockForm` such that it is unique within the generated page. Since the portlet is not aware whether names such as this are unique within an aggregated page, portlets use this token to let the Consumer generate a unique name.
5. Since the Producer is requesting the Consumer to rewrite URLs and namespaces, WSRP terms this approach of URL generation as "consumer rewriting". Refer to Section 10.2.1 of the WSRP 1.0 Specification for rules governing consumer rewriting. Using these rules, C Inc might rewrite the markup fragment into what is shown below.

```
<form method="post"
```

```
  action="financePage?portletID=portfolioManager&type=blockingAction"
  id="portfolioManager_1_stockForm">
  <table border="0" width="100%">
    <tr>
      <td>Enter Stock Symbol</td>
      <td><input name="symbol"></td>
    </tr>
    <tr>
      <td><input type="submit" value="Submit"></td>
    </tr>
  </table>
</form>
```

In this example, the Consumer rewrote the action URL to refer to a finance page containing the portfolio manager portlet. The actual values of the URL and names in the above markup depend on the Consumer's implementation. Another Consumer interacting with P Inc may rewrite the URLs and names differently.

The WSRP specification specifies an alternative form of URL generation called as "producer-writing". Producer writing involves URL templates and a namespace prefix that the Consumer sends to the Producer. Instead of using tokens, Producer uses the templates and the namespace prefix to create URLs and names in the markup. For more details of this approach, refer to Section 10.2.2 of the WSRP 1.0 Specification.

5.2.2 Perform Blocking Interaction

1. Consumers can use the `performBlockingInteraction` operation to send user interactions to the Producer. During this operation, portlets can process user interactions while letting the Producer affect their state. Note that the scope of the `getMarkup` operation is limited to generating markup for a portlet without affecting the current state of the portlet.
2. When a user interacts with a portlet (e.g. by submitting a form), the Consumer uses the `performBlockingInteraction` to send the submitted data to the Producer. During the course of this operation, any/all of the following may happen.
3. Portlets can access and process the request data during a `performBlockingInteraction` request.
4. Portlets can change their navigational state.
5. Portlets can make persistent changes to the state of the portlet.
6. Portlets can ask the Consumer to redirect the user to arbitrary URLs.
7. Portlets can change their current mode and/or window state.
8. The Producer can choose to generate and return the portlet's markup as an optimization.

Let us extend Scenario 4 to let the user cause an interaction affecting the portlet's state.

The user enters a value of "PINC" for the stock symbol in the form displayed by C Inc for the portfolio manager portlet, and submits the form.

C Inc collects the form data from the incoming HTTP request, and sends an interaction request to P Inc.

The portfolio manager portlet stores the symbol and returns.

C Inc then sends a `getMarkup` request to P Inc for an updated markup of the portlet. The portlet determines what symbols it is generating markup for with this user and returns markup showing the value of the stock entered by the user along with any other symbols it had already been showing this user.

After the user submits the form, C Inc sends a `performBlockingInteraction` request to P Inc so that the portfolio manager portlet can process the user interaction. The `performBlockingInteraction` request contains all the data elements contained in the earlier `getMarkup` request, plus an additional `InteractionParams` element. This element includes the form data submitted by the user.

In this example, this portlet is capable of processing only one user interaction. However, if this portlet is capable of processing more interactions, it may add an `interactionState` parameter to the action URL. This parameter can describe the kind of user interaction associated with the request. For example, if this portlet can also delete a stock symbol from the list of symbols, it could add an `interactionState` parameter such as "deleteSymbol". The actual value of the state is opaque and is implementation specific.

Producers or portlets can embed arbitrary state within URLs as interaction state. When a user activates a URL with interaction state, Consumers extract the interaction state from the HTTP request, and include it in the `performBlockingInteraction` request to process the interaction. Interaction state is similar to form parameters except for the difference that Producers or portlets pre-populate interaction state in URLs.

The following message shows a `performBlockingInteraction` request message as it would be generated if the form from our preceding `getMarkup` is submitted by the user to the Consumer requesting the PINC stock symbol. Note that the `portletStateChange` field has been set to `readOnly` to indicate that a state change is not acceptable to the Consumer. We will discuss the purpose of this field later in this section.

The `runtimeContext` element of this request also includes the `sessionID` previously returned by C Inc.

```
<urn:performBlockingInteraction xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:portletContext>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
  </urn:portletContext>
  <urn:runtimeContext>
    <urn:userAuthentication>wsrp:password</urn:userAuthentication>
    <urn:sessionID>sessionID_1</urn:sessionID>
  </urn:runtimeContext>
  <urn:userContext xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
  <urn:markupParams>
    <urn:secureClientCommunication>false</urn:secureClientCommunication>
    <urn:locales>en-US</urn:locales>
    <urn:mimeTypes>text/html</urn:mimeTypes>
    <urn:mode>wsrp:view</urn:mode>
    <urn:windowState>wsrp:normal</urn:windowState>
    <urn:clientData>
      <urn:userAgent>Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.6)
Gecko/20040206 Firefox/1.0</urn:userAgent>
    </urn:clientData>
    <urn:markupCharacterSets>UTF-8</urn:markupCharacterSets>
  </urn:markupParams>
  <urn:interactionParams>
    <urn:portletStateChange>readOnly</urn:portletStateChange>
    <urn:formParameters name="symbol">
      <urn:value>PINC</urn:value>
    </urn:formParameters>
  </urn:interactionParams>
```

```
</urn:performBlockingInteraction>
```

Message 14: Blocking Interaction Request

To this request, P Inc responds with the message shown below.

```
<urn:performBlockingInteractionResponse
xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:updateResponse>
    <urn:navigationalState>symbol=PINC</urn:navigationalState>
  </urn:updateResponse>
</urn:performBlockingInteractionResponse>
```

Message 15: Blocking Interaction Response

The `UpdateResponse` element of this response contains new `navigationalState` created by P Inc. This navigational state represents state that P Inc expects C Inc to return with subsequent `getMarkup` requests. In this specific example, P Inc created a string that contains the name of the stock symbol and returned it as the navigational state. When C Inc returns this state with a future `getMarkup` request, P Inc extracts the name of the stock symbol from the navigational state, looks up for the value of the stock symbol, and generates a markup fragment that shows the name and value of the stock symbol. As far as the Consumer is concerned, the `navigationalState` is completely opaque.

Navigational state is similar to a query string on a standard URL. Producers can express the result of processing the interaction as the `navigationalState` and use this state to generate markup as many times as requested by the Consumer. For instance, if a Consumer sends a `performBlockingInteraction` request with the data necessary to create a purchase order in a database, the Producer could return a reference to the primary key of the purchase order created as navigational state, and use that state in subsequent `getMarkup` requests to show pertinent information of the purchase order created.

Consumers may embed the `navigationalState` (or a reference to the `navigationalState`) in portlet URLs so that users can bookmark pages including the navigational state of each portlet. When a user activates such a bookmarked URL, the Consumer extracts the `navigationalState` and sends it to the Producer to get similar markup as was obtained at the time of book marking.

Depending on the results of processing the `performBlockingInteraction` request, the `performBlockingInteractionResponse` could include the following:

1. An updated `portletContext` with a `portletHandle` and/or `portletState` element. In this case, the Consumer prevented this by setting the `portletStateChange` element to a value of `readOnly`.
2. A sessionContext with a new `sessionID`. Consumer is required to send this `sessionID` with future `getMarkup` or `performBlockingInteraction` requests.
3. New window state and/or mode.
4. A redirectURL in place of the `updateResponse` element. The value of this element is an absolute URL that the Consumer is required to direct the user to.
5. A `markupContext` element with the portlet markup. Note that to let the Producer generate this, the `performBlockingInteraction` includes the full `MarkupParams` structure.

When a Producer returns `markupContext` in the `performBlockingInteractionResponse`, Consumers can avoid calling `getMarkup` and use this markup instead, presuming it honored any requested changes in mode or window state. Consumers can also discard this `markupContext`, and send a usual `getMarkup` request for the markup.

The following figure shows the overall sequence of `getMarkup` and `performBlockingInteraction` requests.

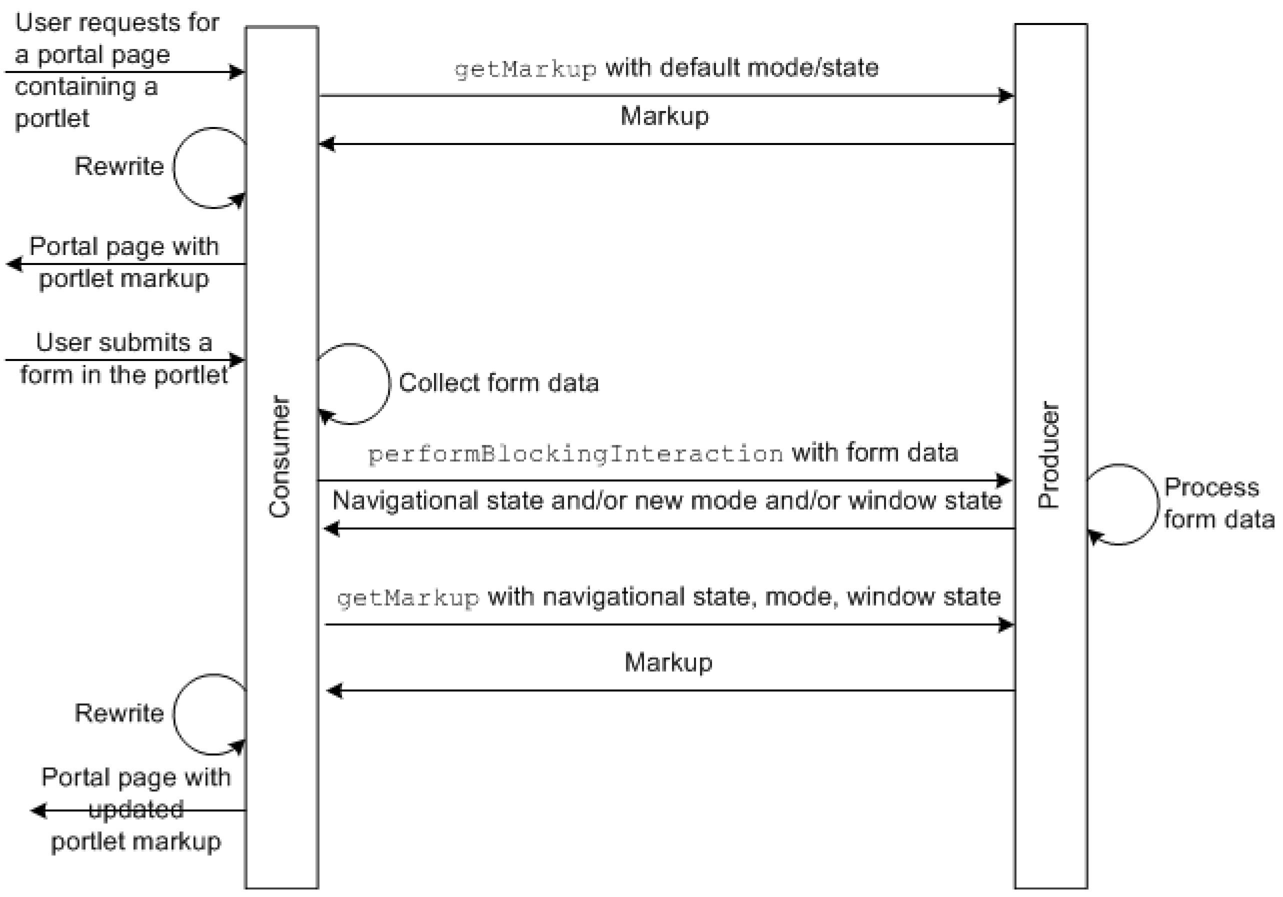
User requests for a portal page containing a portlet
Rewrite
Portal page with portlet markup
User submits a form in the portlet
Consumer
Collect form data
getMarkup with default mode/state
Markup
performBlockingInteraction with form data
Navigational state and/or new mode and/or window state
Producer
Process form data
getMarkup with navigational state, mode, window state
Markup
Rewrite
Portal page with updated portlet markup

Figure 3: Two Step Protocol

In this two-step protocol, the following key differences between `getMarkup` and `performBlockingInteraction` are worth noting.

1. Consumers send `performBlockingInteraction` requests only when a user interacts with a portlet URL specifying a value for url-type of `blockingAction` or secureBlockingAction. On the other hand, Consumers send `getMarkup` requests only whenever the portlet's markup is required or when the url-type is render or secureRender. For example, a simple browser reload of a Consumer's aggregated page or user interactions with other portlets on aggregated page may cause `getMarkup` invocations without any `performBlockingInteraction` invocation.
2. Apart from a new sessionContext, Producers cannot return any state changes in the response to a `getMarkup` request. However, in the case of `performBlockingInteraction` requests, Producers can return such changes to the Consumer.
3. Although Producers cannot reflect state changes to the Consumer, portlets can still make transient/persistent state changes internally. However, Producers must be prepared to process any number of `getMarkup` requests without negative effects. In particular, a portlet making transient and/or persistent changes to its state during markup generation must be prepared to generate markup any number of times.

5.2.3 State Changes and Implicit Cloning

While processing a `performBlockingInteraction` request, if allowed by the Consumer, the Producer can clone the portlet, and return a `portletContext` with a new `portletHandle` and/or `portletState`. This is often called implicit cloning because the Consumer did not directly ask the Producer to clone the portlet, rather the Consumer indicated that under certain circumstances the Producer was to generate a clone and return it. For the Consumer, the new `portletContext` replaces the current `portletContext` for this user.

To illustrate how such implicit cloning may occur, consider the following variation to Scenario 5.

C Inc allows several of its users access the portfolioManager portlet.

One of the users accessing this portlet enters a value of "PINC" for the stock symbol in the form displayed by C Inc for the portfolio manager portlet, and submits the form.

C Inc collects the form data from the incoming HTTP request, and sends an interaction request to P Inc.

The portlet adds this symbol to the user's preferred symbols in a database and returns.

Scenario 6: Perform Interaction with Implicit Cloning

The portlet can use its `portletHandle` as a primary key for storing the list of symbols. However, before letting the portlet add the stock symbol to the list of preferred symbols, P Inc should ensure that the `portletHandle` of this portlet is specific to the current user and is not shared with other users accessing the same portlet. Since P Inc offered this portlet with a `portletHandle` portfolioManager via its service description, there may be several Consumers (or several users from the same Consumer) using this portlet with the same `portletHandle`. In order to avoid sharing the list with other users, P Inc must first create a new `portletHandle` and let the portlet store the list against the new `portletHandle`.

However, P Inc does not know whether C Inc allows several users to access the same portlet or not. That is, without the Consumer supplying additional information, P Inc cannot determine if it must clone the portlet before letting the portlet store the list of symbols. C Inc must therefore inform P Inc that, in case the portlet is trying to make state changes, P Inc must first clone the portlet.

This scenario illustrates one of the ways implicit cloning may occur. Other possibilities depend on how a Producer associates persistent state with portlets.

When a Producer implicitly clones a portlet, it returns the new `portletHandle` (along with `portletState`, if it is not capable of storing state persistently) to the Consumer. However, Consumers may not always be ready to accept a new a `portletHandle` or `portletState`. For instance, the Consumer may not have persistent storage capabilities. Or, the Consumer may not want to allow a given user affect the persistent state of a portlet as the user lacks adequate privileges. To account for these situations, WSRP specifies a `portletStateChange` element in the `performBlockingInteraction` request. Consumer can supply one of the following values for this element:

1. `readOnly`: This value indicates that the Producer is not allowed to return a new `portletHandle` and/or `portletState`. If the portlet tries to update persistent state that may cause a new `portletHandle` or `portletState`, the Producer will return a `PortletStateChangeRequired` fault to the Consumer.
2. `cloneBeforeWrite`: This value indicates that the Producer must first clone the portlet before attempting to make persistent state changes.
3. `readWrite`: This value indicates that the Producer can make persistent state changes without cloning the portlet.

Typically, Consumers capable of accepting implicit cloning initially send a value of `cloneBeforeWrite`, and replace it with `readWrite` once implicit cloning occurs. Producers are required to depend on the Consumer to properly indicate whether or not the user is allowed to update the persistent state for the current `portletHandle`.

The following figure illustrates the persistent lifecycle of a portlet caused by implicit cloning.

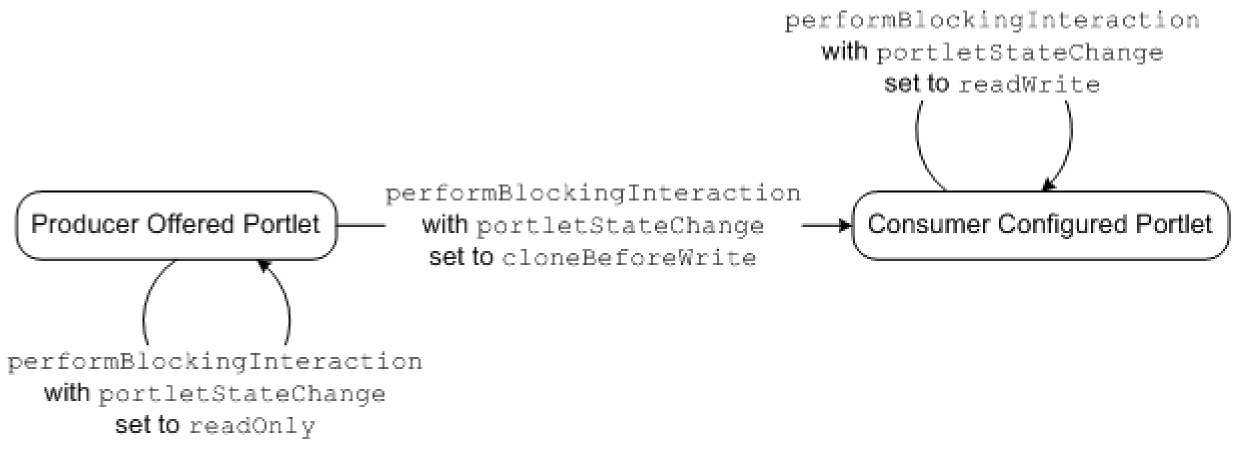

Figure 4: Persistent Lifecycle of Portlets with Implicit Cloning

In this figure, rounded rectangles show two of the states in the persistent lifecycle of a portlet. The arrows between these states show transitions between states.

1. Producer-Offered Portlet: A Producer-offered portlet is one that is described in the `getServiceDescriptionResponse` (e.g. as in Message 2). In our sample scenario, the portfolio manager portlet with portfolioManager as the `portletHandle` is a producer-offered portlet.
2. Consumer-Configured Portlet: A Producer can create a consumer-configured implicitly, when the Consumer sends a value of `cloneBeforeWrite` for `portletStateChange` with the `performBlockingInteraction` request.

In Section 6, we will discuss an explicit method of cloning and destroying portlets.

5.2.4 Initialize Cookies

Between typical browser-server interactions, web servers use cookies to set state to the browser, which the browser returns with subsequent requests. RFC 2965 [8] describes the rules governing cookies.

In the case of WSRP, Producers may set cookies on responses to Consumers, requiring the Consumers to return those cookies with future `getMarkup` and `performBlockingInteraction` requests. Here is a sample scenario that motivates the use of cookies.

P Inc uses a load-balancing cluster of Producers for handling Consumers' requests. The portlets deployed on these Producers manage some transient state in memory. For any one sequence of interactions on behalf of a user, P Inc requires the load balancer to direct a given Consumer's requests for any particular portlet to a given Producer in the cluster.

P Inc would like to use HTTP cookies to achieve this load-balancing scheme.

Scenario 7: Initializing Cookies

P Inc can use the `requiresInitCookie` element in its service description (refer to Section 5.1.18 of the WSRP 1.0 Specification for the allowed values) to inform Consumers of this need and C Inc then uses the `initCookie` operation of the markup interface to implement the scenario.

When C Inc sends an `initCookie` request, P Inc can set HTTP cookies with the response, and require C Inc return those cookies with `getMarkup` and `performBlockingInteraction` requests.

For example, consider that P Inc sets the value of `requiresInitCookie` to any value other than `none` in its service description. C Inc then sends the following `initCookie` request to P Inc.

```
<urn:initCookie xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
</urn:initCookie>
```

Message 16: Init Cookie Request

In return, P Inc sends the following response along with `Set-Cookie` headers (at the HTTP transport level). In the current scenario, the value of the cookie may contain some information to let the load-balancing mechanism identity the Producer instance in the cluster.

```
<urn:initCookieResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types"/>
```

Message 17: Init Cookie Response

C Inc collects these cookies, and resends them with subsequent `getMarkup` and `performBlockingInteraction` requests.

5.2.5 Release Sessions

In Message 13, we saw that the Producer, P Inc. initialized a session for the portfolio manager portlet, and returned a reference to that session as a `sessionID` in its response. The Producer and/or the portlet may also be managing in-memory state in those sessions. As a Consumer interacts with a Producer for different portlets on behalf of a given user, the Producer may initialize sessions for these portlets, and return session IDs of those sessions. Since the Consumer is required to supply these IDs to the Producer on subsequent invocations, the Consumer will have to store those temporarily, for example, in the user's session on the Consumer itself.

What happens if the user stops interacting with the Consumer, or the user's session on the Consumer has timed out? In these cases, the sessions on the Producer will remain alive until they timeout naturally. To let the Producer reclaim storage consumed by such sessions in a timely manner, the Consumer can send a `releaseSessions` request to the Producer to inform it that those sessions will not be referenced by future invocations.

In our scenario, C Inc can send the following request to P Inc to release the session created for the portfolio manager portlet.

```
<urn:releaseSessions xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:sessionIDs>sessionID_1</urn:sessionIDs>
</urn:releaseSessions>
```

Message 18: Release Sessions Request

The Consumer can add several session IDs in this request. To this request, P Inc terminates the session, and returns the following response. The response merely indicates that the Producer has released the session, and the Consumer can no longer use those session IDs.

```
<urn:releaseSessionsResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types"/>
```

Message 19: Release Sessions Response

While Producers are required to support the `releaseSessions` operation, Consumers may or may not send a `releaseSessions` request to the Producer. Therefore, we recommend that Producers expire sessions after meaningful time interval or inactivity interval, and not depend on the `releaseSessions` operation alone.

5.3 Modes and Window States

Consumers can use modes and window states to influence the behavior and markup of a portlet.

Portlets can expose their functionality with different modes, with each mode catering to a particular function. For example, a portlet can provide its default functionality in one mode (e.g. `wsrp:view` mode) and provide customization functionality under a different mode (e.g. `wsrp:edit` mode). WSRP 1.0 Specification specifies `wsrp:view` , `wsrp:edit` , `wsrp:help` , and `wsrp:preview` modes as standard modes that portlets can support.

Window states, on the other hand, let Consumers indicate how much markup a portlet should generate. WSRP 1.0 Specification specifies `wsrp:normal` , `wsrp:minimized` , `wsrp:maximized` , and `wsrp:solo` as standard window states. Portlets could generate different length/style of markup in each of these window states. For example, a portlet could generate its complete view (with more markup, images etc) in `wsrp:maximized` window state, while generating no displayable markup in `wsrp:minimized` state.

In addition to these standard modes and window states, portlets could offer markup in additional modes (known as custom modes) and window states (known as custom window states). Producers advertise the modes supported by a portlet in the description of the portlet. Consumers typically provide decorations (such as a title bar with buttons) to let users request portlet markup in various modes and window states. WSRP 1.0 Specification requires that portlets support `wsrp:view` mode and `wsrp:normal` window state so that Consumers are able to obtain markup in at least one mode and window state they support.

Portlets could request a change to the current mode and/or window state while processing a `performBlockingInteraction` by returning the requested new values within the `performBlockingInteractionResponse` . Consumers are expected (although, not required) to honor such mode and window state changes. If a Consumer does not understand a portlet's mode or window state (including custom modes and custom window states), the Consumer is unlikely to request markup using that mode or window state. In such cases, the Consumer can request for markup in one of the known modes and window states.

A Consumer could have its own set of custom modes and window states, and attempt to request a portlet render in one of these. If the portlet does not comprehend the requested mode or window state, the Producer returns an appropriate fault message.

5.4 CSS Portlet Classes

HTML 4.01 states: “Since style sheets are now the preferred way to specify a document's presentation, the presentational attributes of BODY have been deprecated.” Accordingly, WSRP 1.0 has adopted an initial basic set of CSS classes designed to provide a standard set of display options for portlets. For a list with tables of these portlet classes, see Section 10.6 of WSRP 1.0 specification. Use of CSS portlet classes are optional and only appropriate for those markup types supporting CSS.

5.5 Caching of Markup

WSRP allows the Consumer to cache markup fragments returned by the Producer. This enables the Consumer to avoid calling `getMarkup` again to obtain the same markup fragments that the Producer had returned previously and had indicated as cacheable. In addition to improved performance at the Consumer, caching makes the Producer more efficient since the Producer does not have to regenerate identical markup fragments across a series of requests. Note that the Consumer must take into account the `MarkupParams` structure that the Consumer sent to compute any key used to locate cached markup fragments in the Consumer’s caching mechanism.

The presence of a valid (i.e., non-null) `CacheControl` in the `MarkupContext` sent by the Producer when returning markup is the hint given by the Producer to the Consumer that it may choose to cache the returned markup for subsequent invocations. The `CacheControl` structure includes information such as the duration that the cached markup is valid for, the user scope of the markup (e.g. whether the Consumer can share the markup for other users as well), as well as a tag that the Consumer can use to send to the Producer to validate if the cached markup can still be used by the Consumer even after the expiry of the cache.

To illustrate how caching works, let us revisit Scenario 4, wherein C Inc made a request to P Inc for the initial markup of the portfolio manager portlet.

Here is the response from P Inc.

```
<urn:getMarkupResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:markupContext>
    <urn:mimeType>text/html; charset=UTF-8</urn:mimeType>
    <urn:markupString><![CDATA[
      <form method=”post”
        action="wsrp_rewrite?wsrp-urlType=blockingAction/wsrp_rewrite"
        id="wsrp_rewrite_stockForm">
        <table border="0" width="100%">
          <tr>
            <td>Enter Stock Symbol</td>
            <td><input name="symbol"></td>
          </tr>
          <tr>
            <td><input type="submit" value="Submit"></td>
          </tr>
        </table>
      </form>
    ]]></urn:markupString>
    <urn:locale>en-US</urn:locale>
    <urn:requiresUrlRewriting>true</urn:requiresUrlRewriting>
    <urn:cacheControl>
      <urn:expires>60</urn:expires>
      <urn:userScope>wsrp:perUser</urn:userScope>
      <urn:validateTag>portfolioManagerPValidateTag</urn:validateTag>
    </urn:cacheControl>
    <urn:preferredTitle>Portfolio Manager</urn:preferredTitle>
  </urn:markupContext>
  <urn:sessionContext>
    <urn:sessionID>sessionID_1</urn:sessionID>
    <urn:expires>300</urn:expires>
  </urn:sessionContext>
</urn:getMarkupResponse>
```

Message 20: Get Markup response that includes caching information

This message is similar to Message 13, except for the parts shown in bold. In this response, P Inc returned a `CacheControl` element as part of the `MarkupContext` that indicates to C Inc that the markup fragment returned is cacheable.

In this particular response, the `CacheControl` structure includes the following elements:

1. `expires`: This field indicates that the markup fragment referenced by this cache control is valid for 60 seconds (counting from point in time when the markup was returned).
2. `userScope`: A value of `wsrp:perUser` specifies that the markup is specific to the `userContext` for which it was generated.
3. `validateTag`: The value `portfolioManagerPValidateTag` is for the purpose of the Consumer to verify with the Producer if this particular cached markup fragment is still valid even after it has expired (i.e. in the situation whereby, even though the cache had expired, calling `getMarkup` would result in the same markup fragment being returned).

Next, let us suppose that the user at C Inc refreshed the page in the browser. In the absence of any caching, this would have caused the Consumer to get the markup for the portlet from the Producer again. However, since the markup that was just returned did contain a valid `CacheControl`, the Consumer can use the cached markup.

Now let us explore the scenario whereby the user refreshed the page after the expiry of the cached markup (i.e., after 60 seconds). Since the Producer returned a validateTag element as part of the `CacheControl`, the Consumer has the option of sending the value of that element back to the Producer to determine if it can still reuse that cached markup.

Here is the new `getMarkup` request from C Inc to P Inc.

```
<urn:getMarkup xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:portletContext>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
  </urn:portletContext>
  <urn:runtimeContext>
    <urn:userAuthentication>wsrp:password</urn:userAuthentication>
  </urn:runtimeContext>
  <urn:userContext xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
  <urn:markupParams>
    <urn:secureClientCommunication>false</urn:secureClientCommunication>
    <urn:locales>en-US</urn:locales>
    <urn:mimeTypes>text/html</urn:mimeTypes>
    <urn:mode>wsrp:view</urn:mode>
    <urn:windowState>wsrp:normal</urn:windowState>
    <urn:clientData>
      <urn:userAgent>Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.6)
Gecko/20040206 Firefox/1.0</urn:userAgent>
    </urn:clientData>
    <urn:markupCharacterSets>UTF-8</urn:markupCharacterSets>
    <urn:validateTag>portfolioManagerPValidateTag</urn:validateTag>
  </urn:markupParams>
</urn:getMarkup>
```

Message 21: Get Markup Request with a validate tag

The above request from the C Inc to P Inc is similar to Message 12 except for the line shown in bold. The `MarkupParams` structure now also contains a validateTag element with the value `portfolioManagerPValidateTag` that P Inc returned earlier as part of the `CacheControl` structure. The Consumer is supplying this as a means for the portfolio manager portlet to avoid generating new markup if it can valid this tag.

Let us assume that the markup is indeed still valid. P Inc responds to the `getMarkup` request as shown below.

```
<urn:getMarkupResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:markupContext>
    <urn:useCachedMarkup>true</urn:useCachedMarkup>
    <urn:cacheControl>
      <urn:expires>180</urn:expires>
      <urn:userScope>wsrp:perUser</urn:userScope>
      <urn:validateTag>portfolioManagerPValidateTag</urn:validateTag>
    </urn:cacheControl>
  </urn:markupContext>
  <urn:sessionContext>
    <urn:sessionID>sessionID_1</urn:sessionID>
    <urn:expires>300</urn:expires>
  </urn:sessionContext>
</urn:getMarkupResponse>
```

Message 22: Get Markup Response that includes a `CacheControl` structure

In the response above, the Producer indicates that the cached markup fragment is still valid for this request by setting the `useCachedMarkup` to `true`. Note that the Producer intentionally omits the `markupString` field from the response. Also, note that it is mandatory for the Producer to send back a new `cacheControl` structure to the Consumer to indicate a new expiry field for the given markup fragment. The Consumer should replace this with any other value it had stored earlier.

Lastly, with respect to caching invalidation, we recommend Consumers discard any cached markup for the given portlet once they invoke `performBlockingInteraction` since the operation may result in the cached markup becoming invalid. Future versions of WSRP may provide a means for a portlet to indicate that the markup cached by the Consumer is no longer valid.

5.6 Recommendations

The markup interface deals with markup, user interactions, and the state associated. Due to the complexities associated with these, Producer and Consumer implementers must take into account a variety of issues and choices. During the lifetime of a portlet, the markup interface gets more heavily used than other interfaces, and hence implementers must take into account performance and scalability issues as well. In addition, Producers invoke portlet(s) during this interface, portlet developers must also be aware of certain intricacies. In this section, we present some guidelines for implementers and portlet developers.

The first thing to note about the markup interface is the two-phase protocol. Producers must capture the result of an interaction as navigational state or some other form transient/persistent state and be prepared to generate markup any number of times with the help of that state.

Producers can use `navigationalState` or sessions (or both) to manage transient state. However, `navigationalState` has some advantages over using sessions. In particular, Producers/Consumers can embed `navigationalState` in URLs in markup returned to end users. Users can therefore bookmark URLs and be able to view the same/similar content at a future time. Since sessions usually have limited lifetime, after the Producer terminates a session, users may not be able to view same/similar content at a future time.

Use of `navigationalState` to represent transient state also improves cacheability of markup by Consumers. Consumers can use the `navigationalState` as part of the cache key for caching the markup, and serve cached markup whenever the same `navigationalState` is found for a given `getMarkup` request.

Another point to consider is the network traffic. In order to reduce network overhead, we recommend the following practices:

1. If a Producer can return markup with its response to `performBlockingInteraction`, Consumers can avoid invoking the `getMarkup` request for the same portlet on the same Producer, saving one network roundtrip.
2. We also encourage Producers and Consumers to cache markup whenever possible. Particularly, markup caching by Consumers can significantly reduce network traffic.
3. If a Consumer is aggregating several portlets from several Producers, the Consumer should try to invoke `getMarkup` requests concurrently instead of serially. Provided the Consumer has sufficient network bandwidth, this approach will help improve responsiveness of the Consumer as far as end users are concerned.

Another issue for consideration is URL generation. Producers must carefully evaluate their portlets and usage to support one of the forms of URL generation. In general, URL template based URL generation offers better performance when the markup is personalized for each user and the Consumer is able to supply URL templates and not post-process the markup returned from the Producer. On the other hand, Consumer URL rewriting has the advantage of cacheability. With Consumer URL rewriting, the markup returned by the Producer does not include references to the Consumer, and therefore the Producer can cache the markup and serve the cached markup to several Consumers.

Although Producers can choose to support either Producer writing or Consumer rewriting, Consumers must be prepared to support both so that the Consumer can easily work with diverse Producer implementations.

6 Portlet Management Interface

6.1 Introduction

The purpose of the portlet management interface is to let Consumers manage the persistent state and lifecycle of portlets explicitly.

6.1.1 Portlet Persistent State

In addition to the transient state (such as navigational state) of portlets we discussed in the previous section, portlets can have persistent state as well. WSRP allows Producers to expose a transparent view on such persistent state as properties. Portlet properties are data associated with a portlet. Using the portlet management interface, Consumers can access and change those properties.

An example of a portlet property is the list of stock symbols for the portfolio manager portlet.

While this portlet encapsulates the functionality necessary to manage portfolios, the portlet may declare the list of the stock symbols as a property. Each Consumer can use the portlet management interface to clone this portlet for each user, and set the values of the stock symbol property for each user.

Note that the persistent state of producer-offered portlets is not explicitly modifiable by Consumers. However, when a Producer exposes such persistent state via properties, Consumers can use the portlet management interface to create a consumer-configured portlet, and modify the exposed portion of its persistent state explicitly.

6.1.2 Portlet Lifecycle

The Portlet Management interface provides for the following persistent lifecycle of portlets.

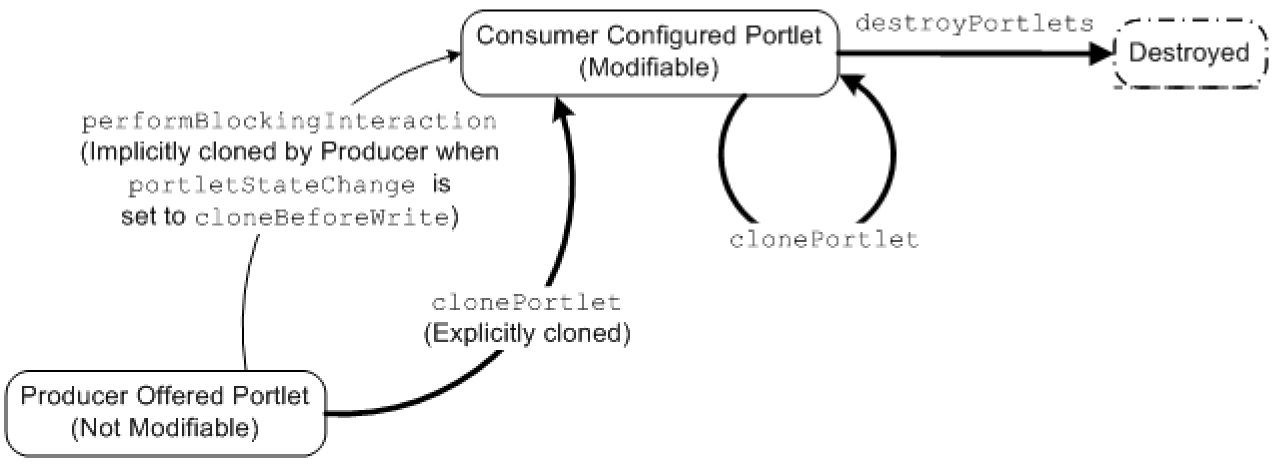
Consumer Configured Portlet (Modifiable)
destroyPortlets
Destroyed
performBlockingInteraction (Implicitly cloned by Producer when portletStateChange is set to cloneBeforeWrite)
clonePortlet
clonePortlet (Explicitly cloned)
Producer Offered Portlet (Not Modifiable)

Figure 5: Lifecycle of Portlets

The lifecycle of portlets consists of three states (shown in rounded rectangles). The arrows between these states indicate transitions between these states. The bold arrows show the transitions that a Consumer can cause explicitly using the portlet management interface.

1. Producer-Offered Portlet: A Producer-offered Portlet is one that is described in the `getServiceDescriptionResponse` (e.g. as in Message 2). Consumers cannot explicitly modify the persistent state of such portlets.
2. Consumer-Configured Portlet: A consumer-configured Portlet can be created in two ways – implicitly, or explicitly. As discussed in Section 5.2.3, a Producer can implicitly create such a portlet during a `performBlockingInteraction` operation when the Consumer sends a value of `cloneBeforeWrite` for the `portletStateChange` flag. A Consumer can explicitly create such a portlet by sending a request to clone the portlet to the Producer. Consumers can change the properties of consumer-configured portlets. Consumers can also clone Consumer-configured portlets.
3. Destroyed Portlet: This is the final state of a consumer-configured portlet. A Consumer can request the Producer to destroy a consumer-configured Portlet. After a Producer destroys a portlet, the Consumer can no longer use it. Note that Consumers cannot destroy producer-offered portlets.

In addition to the operations to manage this lifecycle explicitly, the portlet management interface offers methods to get the descriptions of producer-offered or consumer-configured portlets, and to get/set properties of a portlet.

The portlet Management Interface offers the following operations:

1. `getPortletDescription`: Consumers can invoke this operation to get a description

of a producer-offered or consumer-configured portlet.

2. `getPortletPropertyDescription`: Consumers can invoke this operation to obtain a description of properties (if any) of a portlet. This operation returns the metadata (such as names, and schema types) of properties.
3. `getPortletProperties`: Consumers can invoke this operation to obtain the properties (including their current values) of a producer-offered or a consumer-configured portlet.
4. `clonePortlet`: Consumers can invoke this operation to explicitly clone a portlet, such that any properties associated with the cloned portlet may be modified without affecting the properties of the portlet that it is cloned from. Consumers can clone both producer-offered and consumer-configured portlets.
5. `setPortletProperties`: Consumers can invoke this operation to modify the values of properties of a consumer-configured portlet.
6. `destroyPortlets`: Consumers can explicitly destroy consumer-configured portlets using this operation.

6.2 Descriptions of Operations

6.2.1 Get Portlet Description

While the `getServiceDescription` operation of the service description interface returns descriptions of all portlets offered to a given Consumer, the `getPortletDescription` operation returns the description of a single portlet. This operation serves two purposes:

1. Given the `portletHandle` of a portlet, the Producer returns the description of a producer-offered or a consumer-configured portlet.
2. The Producer can optionally tailor the description of a portlet based on the user context and registration context supplied by the Consumer.

For producer-offered portlets, in response to a `getPortletDescription` request, Producers are free to return the same portlet description as is returned in the `getServiceDescriptionResponse`.

Let us consider the following scenario, and discuss the semantics of this operation.

C Inc provides a page for its users to view the description of the portfolio manager portlet.

C Inc sends a request to P Inc to get the description of the portfolio manager portlet, and uses the returned description to display a user-friendly page with the description of the portlet.

Scenario 8: Get Portlet Description

To get the description of the portfolio manager portlet, C Inc has two options. If the `portletHandle` of the portlet is producer-offered, C Inc can send a `getServiceDescription` request and search the response for a description of the portfolio manager portlet. Alternatively, it can send a `getPortletDescription` request to get a description of the portlet. Some advantages of the later option is that C Inc can use the same operation for producer-offered as well as consumer-configured portlets and that the returned data is likely to represent any filtering that P Inc does for the user.

C Inc. sends the following message to obtain the description of the portfolio manager portlet.

```
<urn:getPortletDescription xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
```

```
    <urn:registrationHandle>CincRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:portletContext>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
  </urn:portletContext>
  <urn:userContext xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
  <urn:desiredLocales>en</urn:desiredLocales>
</urn:getPortletDescription>
```

Message 23: Get Portlet Description Request

In response, P Inc sends the following message with the description of the portlet:

```
<urn:portletDescription xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:portletHandle>portfolioManager</urn:portletHandle>
  <urn:markupTypes>
    <urn:mimeType>text/html</urn:mimeType>
    <urn:modes>wsrp:view</urn:modes>
    <urn:windowStates>wsrp:normal</urn:windowStates>
    <urn:locales>en</urn:locales>
  </urn:markupTypes>
  <urn:description xml:lang="en">
    <urn:value>Manages portfolios</urn:value>
  </urn:description>
  <urn:title xml:lang="en">
    <urn:value>Title</urn:value>
  </urn:title>
</urn:portletDescription>
```

Message 24: Get Portlet Description Response

The `portletDescription` returned in this message is the same as the one returned by the `getServiceDescription` operation of the service description interface shown in Message 7. Some advanced Producer implementations may tailor (for example, not advertise certain window states or modes) the returned description based on the `registrationContext` and userContext supplied by the Consumer.

6.2.2 Get Portlet Property Description

The `getPortletPropertyDescription` operation returns a description and metadata of properties of a given portlet. Consumers can use this metadata to design user interfaces or applications to view/modify portlet properties. Note that this method does not return the values of properties.

Let us consider the following scenario.

P Inc exposes the list of stock symbols and the refresh interval as properties that Consumers of the portfolio manager portlet can modify.

C Inc would like to setup a page to let users/administrators view and modify these properties.

To setup such a page, C Inc must first know the descriptions and the data types of properties of the portfolio manager portlet.

Scenario 9: Get Portlet Property Description

In order to implement this scenario, C Inc sends a `getPortletPropertyDescription` request to P Inc. Using the returned descriptions and data types of these properties, C Inc designs a page to view/modify the properties.

```
<urn:getPortletPropertyDescription
xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:portletContext>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
  </urn:portletContext>
  <urn:userContext xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
</urn:getPortletPropertyDescription>
```

Message 25: Get Portlet Property Description Request

The portfolio manager portlet has two properties viz., `stockSymbolList` and `refreshInterval`. P Inc therefore returns the following response:

```
<urn:getPortletPropertyDescriptionResponse
xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:modelDescription>
    <urn:propertyDescriptions type="urn:stringValue" name="stockSymbolList">
      <urn:label xml:lang="en">
        <urn:value>Stock Symbols</urn:value>
      </urn:label>
    </urn:propertyDescriptions>
    <urn:propertyDescriptions type="xs:int" name="refreshInterval">
      <urn:label xml:lang="en">
        <urn:value>Refresh Interval</urn:value>
      </urn:label>
    </urn:propertyDescriptions>
  </urn:modelDescription>
</urn:getPortletPropertyDescriptionResponse>
```

Message 26: Portlet Property Description Response

This response includes two properties – a `stockSymbolList` property of type `xs:string`, and a `refreshInterval` property of type `xs:int`. The names, descriptions, and data types of these properties help C Inc. design a user interface for displaying and entering new values for the properties. In addition to the standard schema types, Producers can use any arbitrary schema types (other than those defined in the `urn:oasis:names:tc:wsrp:v1:types` namespace) to describe portlet properties.

In addition to the type, a Producer may optionally supply a label and a hint. In the above message, labels provide a short description of each property.

6.2.3 Get Portlet Properties

The purpose of `getPortletProperties` operation of the portlet management interface is to return all or some properties of a given portlet. Consumers can use this operation in conjunction with the `getPortletPropertyDescription` operation to show portlet properties to users.

Let us consider the following scenario.

Having obtained the descriptions of portlets, C Inc prepares a page with user interface to display the properties of the portfolio manager portlet.

A user enters this page to view the properties.

Scenario 10: Get Portlet Properties

To implement this scenario, C Inc sends the following message to get the current values of properties of the portfolio manager portlet.

```
<urn:getPortletProperties xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:portletContext>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
  </urn:portletContext>
  <urn:userContext xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
  <urn:names xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
</urn:getPortletProperties>
```

Message 27: Get Portlet Properties Request

In this request, the value of the portlet handle is the same that of the portfolio manager portlet offered in the service description of P Inc.

With the `getPortletProperties` request, the Consumer can optionally indicate if it wants the values of all properties, or only for specific properties. When the names element is set to `nil`, this request implies that the Producer must return values for all properties of this portlet. In case the Consumer is interested only in the values of certain properties, it can specify the names for which it needs values. This option is particularly useful when the portlet has a large number of properties. In the following request, C Inc specifies the `stockSymbolList` property.

```
<urn:getPortletProperties xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:portletContext>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
  </urn:portletContext>
  <urn:userContext xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/
  <urn:names> stockSymbolList</urn:names>
</urn:getPortletProperties>
```

Message 28: Get Portlet Properties Request (For Specific Properties)

The default value of the `stockSymbolList` property is “AMZN” and the default value of the `refreshInterval` field is 180 seconds. For the request in Message 27, P Inc returns the following response with these values.

```
<urn:getPortletPropertiesResponse
xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:properties name=" stockSymbolList">
    <urn:stringValue>AMZN</urn:stringValue>
  </urn:properties>
  <urn:properties name="refreshInterval">
    <xs:int xmlns:xs="http://www.w3.org/2001/XMLSchema">180</xs:int>
  </urn:properties>
</urn:getPortletPropertiesResponse>
```

Message 29: Get Portlet Properties Response

C Inc can now populate the user interface with these values filled in. Note that, as the first property is of type `stringValue`, P Inc returned a `stringValue` element, and for the second property, P Inc returned a namespaced `xs:int` element.

6.2.4 Clone Portlet

As we discussed above, Consumers cannot modify persistent state of producer-offered portlets. In order to be able to change persistent state, Consumers must first let the Producer clone a portlet. In WSRP, Consumers use the `portletHandle` to uniquely refer to a portlet. After cloning, Producers associate a new portletHandle to the portlet. As long as the Consumer or the Producer makes no changes to the cloned portlet, the cloned portlet and the portlet it is cloned from have identical persistent state, and should behave the same.

Once a Consumer clones a portlet, the Consumer can modify its persistent state without affecting the portlet it is cloned from.

Note that cloning does not imply any hierarchical relationship between the cloned portlet and the portlet it is cloned from.

Let us consider the following scenario.

C Inc would like to let each user customize the properties of the portfolio manager portlet. Using the descriptions of the properties and their values, C Inc prepares a page with a user interface to view and modify the values.

Using this user interface, users can update the values of the properties.

Scenario 11: Clone the Portlet

In order to implement this scenario, C Inc must first clone the portfolio manager portlet because it cannot directly modify the properties of a producer-offered portlet.

C Inc sends the following `clonePortlet` request to P Inc.

```
<urn:clonePortlet xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:portletContext>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
  </urn:portletContext>
  <urn:userContext xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
</urn:clonePortlet>
```

Message 30: Clone Portlet Request

P Inc creates a clone of this portlet, and returns the following response with a new portlet handle.

```
<urn:clonePortletResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:portletContext>
    <urn:portletHandle>portfolioManager.1</urn:portletHandle>
  </urn:portletContext>
</urn:clonePortletResponse>
```

Message 31: Clone Portlet Response

The returned portlet handle corresponds to a consumer-configured portlet created explicitly by cloning a producer-offered portlet, and is valid for the duration of the Consumer's registration.

If the Producer is capable of storing state of cloned portlet, the Producer may just return a `portletHandle`.

However, if the Producer is not capable of storing the state of the cloned portlet persistently, the Producer may choose to return such state as `portletState` to the Consumer. In such cases, the Consumer would be responsible for persistently storing the `portletState` along with the `portletHandle` and return these to the Producer in future invocations.

C Inc now uses the new portlet handle during all subsequent requests for the portfolio manager portlet for that user. Note that C Inc may also use the new portlet handle with a `getPortletDescription` request to get a description of the portlet.

6.2.5 Set Portlet Properties

Consumers can use the `setPortletProperties` operation of the portlet management interface to modify the properties of consumer-configured portlets. Note that it is an error to use this operation to attempt to modify the properties of producer-offered portlets.

Referring to our scenario, having cloned the portfolio manager portlet, C Inc can change the persistent state of the cloned portlet. When a user fills in new values for the values of these properties, and submits a form, C Inc uses the `setPortletProperties` operation to let P Inc change properties.

A User enters "AMZN YHOO" for the `stockSymbolList` property and "60" for `refreshInterval` and submits a form.

Scenario 12: Setting Portlet Properties

C Inc sends the following request to P Inc. This request includes a cloned `portletHandle` obtained via the `clonePortlet` request in the previous section, and the new values of the properties.

```
<urn:setPortletProperties xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:portletContext>
    <urn:portletHandle>portfolioManager.1</urn:portletHandle>
  </urn:portletContext>
  <urn:userContext xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"/>
  <urn:propertyList>
    <urn:properties name="stockSymbolList">
      <urn:stringValue>AMZN YHOO</urn:stringValue>
    </urn:properties>
    <urn:properties name="refreshInterval">
      <xs:int xmlns:xs="http://www.w3.org/2001/XMLSchema">60</xs:int>
    </urn:properties>
  </urn:propertyList>
</urn:setPortletProperties>
```

Message 32: Set Portlet Properties Request

P Inc. updates the values of the properties, and returns the following response. Note that the WSRP specification does not allow the Producer to change the `portletHandle` in this response. The purpose of the `portletContext` structure in this response is only to allow the Producer to return any `portletState`.

```
<urn:setPortletPropertiesResponse
xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:portletContext>
    <urn:portletHandle>portfolioManager.1</urn:portletHandle>
  </urn:portletContext>
</urn:setPortletPropertiesResponse>
```

Message 33: Set Portlet Properties Response

If the Producer is unable to store updated portlet properties, it may serialize updated properties and return those as `portletState` in the `setPortletPropertiesResponse`.

When a Consumer receives such `portletState`, it is required to supply the same with `PortletContext` in all future invocations for that portlet.

The following sequence shows the sequence of interactions for setting portlet properties with cloning.

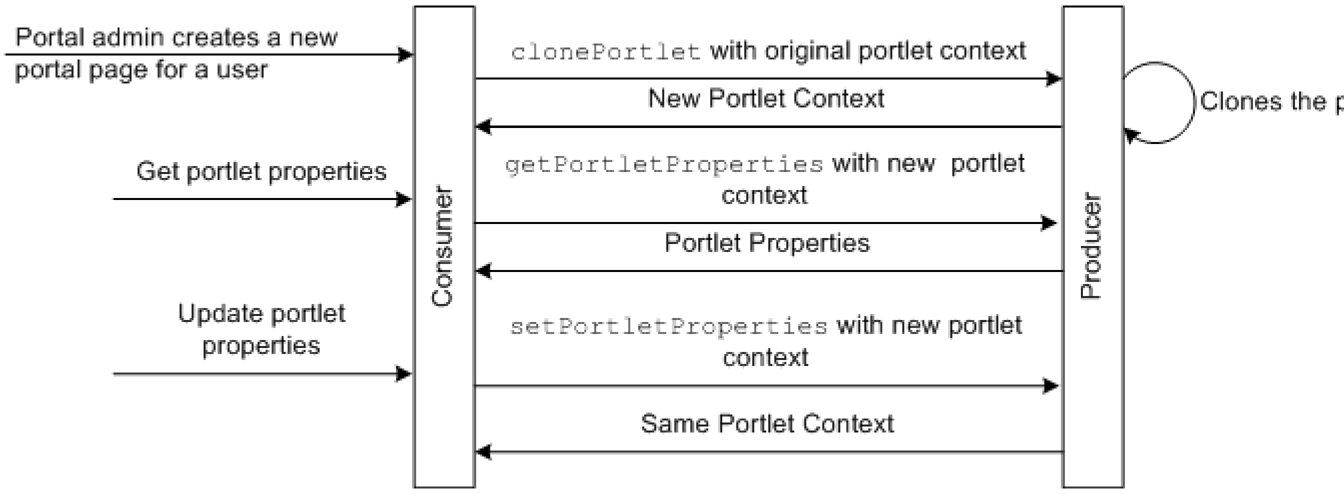
Portal admin creates a new portal page for a user
Get portlet properties
Update portlet properties
Consumer
clonePortlet with original portlet context
New Portlet Context
getPortletProperties with new portlet context
Portlet Properties
setPortletProperties with new portlet context
Same Portlet Context
Producer
Clones the p

Figure 6: Cloning and Setting Portlet Properties

6.2.6 Destroy Portlets

In the above scenarios, the Consumer cloned a producer-offered portlet, so that it can update the persistent state of the portlet. In most cases, Producer implementations will have to store some state in a persistent store such as a relational database. Once the Consumer determines that a consumer-configured portlet is no longer in use, the Consumer should request the Producer to destroy that portlet as this will allow the Producer to clean or archive any stored state.

The `destroyPortlets` operation serves this purpose. It allows a Consumer to request a Producer to destroy one or more consumer-configured portlets. Consider the following scenario.

A user requests C Inc to terminate his user account with C Inc. Since that user’s customizations are no longer required, C Inc sends a request P Inc to destroy portlets cloned for that user.

Scenario 13: Destroy Portlets

In order to destroy portlets, C Inc sends the following request to P Inc.

```
<urn:destroyPortlets xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
```

```
  <urn:registrationContext>
    <urn:registrationHandle>CIncRegnHandle</urn:registrationHandle>
  </urn:registrationContext>
  <urn:portletHandles>portfolioManager.1</urn:portletHandles>
</urn:destroyPortlets>
```

Message 34: Destroy Portlets Request

Note that C Inc may send more than one portlet handle in this request, so that the Producer can destroy several portlets in a single request.

Upon verifying that the portlet handle refers to a consumer–configured portlet, P Inc can delete or archive any persistent state, and return the following response.

```
<urn:destroyPortletsResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types"/>
```

Message 35: Destroy Portlets Response

However, if the Consumer attempts to destroy a producer-offered portlet, or if the Producer fails to destroy a portlet due to some internal failure, the Producer may include the portlets that it failed to destroy and a reason for the failure in the `destroyPortletsResponse`. The following message shows the response from P Inc when it fails to destroy a portlet.

```
<urn:destroyPortletsResponse xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:destroyFailed>
    <urn:portletHandle>portfolioManager.1</urn:portletHandle>
    <urn:reason>Failed to destroy the portlet.</urn:reason>
  </urn:destroyFailed>
</urn:destroyPortletsResponse>
```

Message 36: Destroy Portlets Response When Failed

This response indicates the P Inc failed to destroy the portlet with handle portfolioManager.1.

After the Producer destroys a consumer–configured portlet, it can no longer use the portlet with that handle.

The following sequence illustrates a Consumer aggregating cloned portlets for several users, and destroying the cloned portlets when such portlets are no longer required.

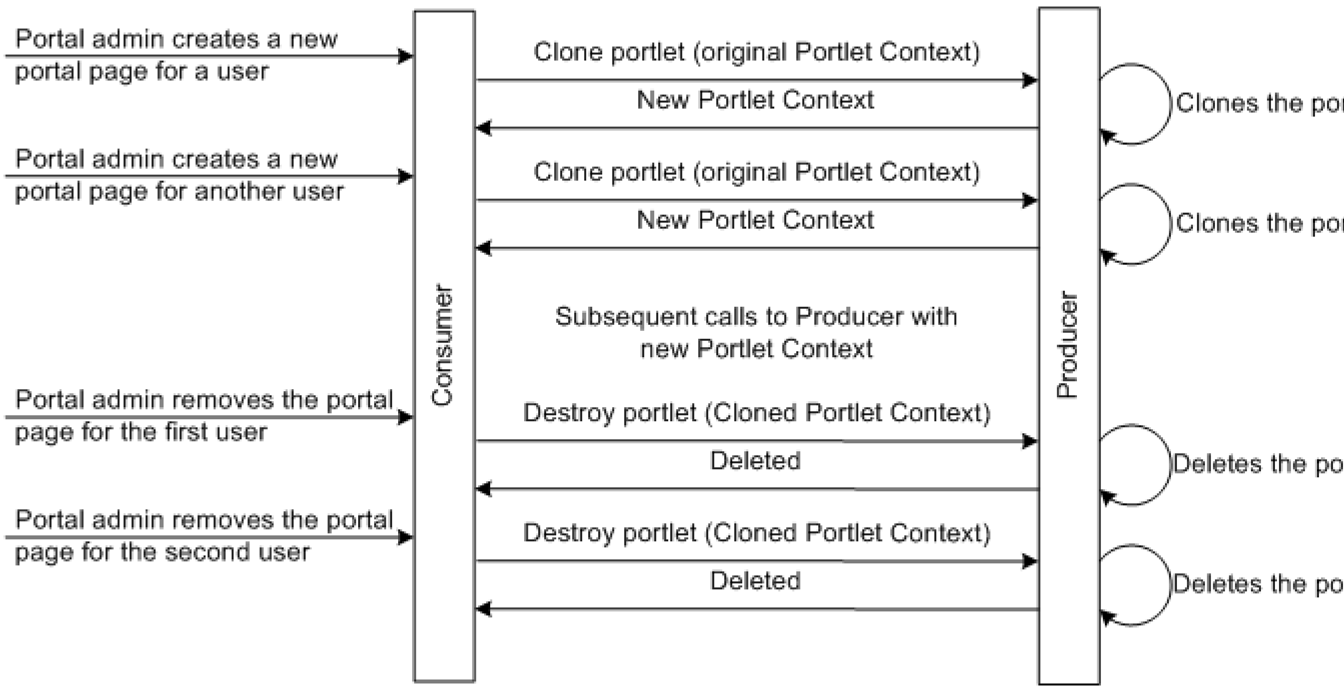
Portal admin creates a new portal page for a user
Clone portlet (original Portlet Context)
New Portlet Context
Clones the po
Portal admin creates a new portal page for another user
Clone portlet (original Portlet Context)
New Portlet Context
Clones the po
Consumer
Subsequent calls to Producer with new Portlet Context
Producer
Portal admin removes the portal page for the first user
Destroy portlet (Cloned Portlet Context)
Deleted
Deletes the po
Portal admin removes the portal page for the second user
Destroy portlet (Cloned Portlet Context)
Deleted
Deletes the po

Figure 7: Cloning and Destroy for Managing Portlet Customizations

6.3 Recommendations

The portlet management interface is an optional interface, and the WSRP specification does not require Producers to implement this interface. If you are implementing a Producer, consider implementing the portlet management interface to allow users/Consumers of your portlets to explicitly manage the lifecycle and customize the properties of portlets. You can thus allow each usage to behave differently while sharing the same implementation (i.e., code) of portlets. To take full advantage of this interface, your portlets must be able to expose some/all of their persistent state as properties. If none of your portlets is capable of doing so, you may not find much value in implementing this interface. Note that the WSRP specification requires support for this interface if portlet clones are ever created.

One of the common choices faced by Producers implementing this interface is whether to expose portlet properties as simple types (such as strings, integers etc) or as complex types. Exposing properties as complex types has the advantage of granularity and typing. However, for Consumers, it may prove to be complex to generate a generic user interface to show/modify complex portlet properties. If a Producer exposes properties as complex types, Consumers may have to generate special purpose user interfaces for each complex type. In order to let Consumers generate a generic user interface, we recommend using simple types.

7 Use Profiles

The purpose of use profiles is to provide a well-defined palette of functionality for Producers and Consumers. By qualifying a Producer or a Consumer implementation with a use profile palette, implementers can indicate the level of functionality supported. WSRP use profiles are non-normative, and should be regarded as general guidelines. Implementers will likely compose their implementations by selecting from a “palette” of functionality, and this section provides some guidance on how to map these use profiles across several functional axes. Refer to [9] for a complete description of use profiles.

WSRP use profiles declare following levels for Producers:

1. Base Level: The Producer implements just the required elements of the WSRP Specification.
2. Simple Level: The Producer implements certain optional/advanced features in addition to whatever is required at the base level.
3. Complex Level: Complex level Producers implement most of the optional/advanced features of the WSRP specification.

The following are the use profiles for Consumers:

1. Base Level: The Consumer implements just the required elements of the WSRP Specification.
2. Simple Level: The Consumer implements certain optional/advanced features in addition to whatever is required at the base level.
3. Medium Level: Medium level Consumers may implement a larger set of the optional/advanced features of the WSRP specification.
4. Complex Level: Complex level Consumers provide extended features (such as custom modes and window states).

The use profiles are intended to simplify the possible combinations of support for optional areas of the WSRP specification. Each profile specifies a certain set of functionality as supported. While an implementation may also support other optional functions, specifying the supported use profile helps customers to compare the support offered by implementations.

A Consumer and Producer have different motivations in achieving higher levels. A Producer need only implement the functionality required by the portlets it is offering. Unless a Consumer knows that it will only be consuming portlets from a Producer of a given level, it should provide all levels so that any portlet can function properly. The Consumer should not assume that there is graceful degradation of functionality if it does not implement certain functionality. For example, if a Consumer does not provide, say, registration, a portlet from a Producer requiring registration will not function at all.

Also note that there is not a one-to one correspondence with Producer Levels and Consumer levels; e.g. the base consumer level is expected to handle the `initCookie` operation, while the base Producer level does not require this operation.

The following table illustrates some common scenarios that Producers and Consumers implement, and provides a mapping of those implementations to user profiles.

Functionalit y/Use Case	**Notes**	**Con sum er Lev el**	**Pr od uce r Le vel**
Implements all required interfaces	The required interfaces are markup and service description interfaces. These interfaces are required so that the Producer and Consumer can offer some minimal level of portlet aggregation.	Bas e	Bas e
Markup Interface			
Producer requires cookie	Base level Consumer must honor the requiresInitCookie element of the service description of a Producer.	Bas e	Si mp

initialization for markup operations			le
Support Consumer-rewriting of URLs	Consumers should at least support consumer rewriting of URLs. In our sample scenario, P Inc and C In rely consumer-rewriting for generating URLs and rewriting names in markup fragments.	Bas e	Bas e
Producer URL Writing	For Producer to be able to create URLs, Consumer submits URL templates. In our sample scenario, P Inc does not support producer-writing of URLs	Co mpl ex	Co mp lex
Support `wsrp:normal` window state and `wsrp:view` mode	Producers must be able to support at least one mode and window state. Normal window state (`wsrp:normal`) and view mode (`wsrp:view`) are the values that Producers and Consumers must support. In our sample scenario, P Inc supports `wsrp:view` mode and `wsrp:normal` window state.	Bas e	Bas e
Support markup types markup	Portlets must be able to support at least one type of markup. In our sample scenario, the portfolio manager portlet offers text/html markup.	Bas e	Bas e
Support Navigational State	Navigational state is one of the basic form of representing transient state of a portlet. In our sample scenario, P Inc includes the user supplied stock symbol and its value as the navigational state for C Inc to return with `getMarkup` requests.	Bas e	Si mp le
Session State	For Producers, managing session state is optional. However, Consumers must support sessions to guarantee a basic level of aggregation of portlets. In our sample scenario, P Inc manages session state.	Bas e	Si mp le
Markup Caching	Producer supplies a `cacheControl` element to indicate Consumer whether it can cache the markup.	Sim ple	Si mp le
Supports Standard Modes	`wsrp:edit`, `wsrp:help`, and `wsrp:preview` modes. P Inc uses standard modes.	Sim ple	Si mp le
Supports Standard Window States	`wsrp:maximized`, `wsrp:minimized` and `wsrp:solo` window states. P Inc uses standard window states	Sim ple	Si mp le
Caching validation	Use the validateTag field of `MarkupParams`	Co mpl	Co mp

		ex	lex
Multiple Markup Types	Portlets could support multiple markup types (e.g. text/html and text/wml). This would allow Consumers to provide portlet aggregation for various devices and non-browser environments.	Complex	Simple
Supports Custom Modes	e.g. a print mode	Complex	Complex
Supports Custom Window States	e.g. a half-page mode	Complex	Complex
Registration Interface			
In-band Registration	Base level Consumers cannot display portlets that require registration. P Inc uses in-band registration.	Simple	Simple
Out-of-band Registration	Complex level producers could allow out-of-band registration by creating a `registrationHandle` by other means than the `register` operation.	Complex	Complex
Portlet Management Interface			
Implicit or explicit cloning of portlets	Base level Consumer and Producer do not support cloning.	Simple	Simple
Grouping of Portlets	Producers could group portlets to allow portlets in a group share transient/persistent state. Refer to Section 3.8 of the WSRP 1.0 Specification for details.	Complex	Complex
Persistent local state	Producers could manage persistent state locally and not return `portletState` or `registrationState`. In our sample scenario, P Inc is capable of managing persistent state locally.	Simple	Complex
Advanced			
Portlet Management Interface	Consumers can use the portlet management interface to manage persistent lifecycle of portlets explicitly. Consumer may create a user interface for property management using this interface.	Complex	Simple

Localization	Producers may be able to supply localized values for resources such as names, descriptions etc. In our scenario, P Inc supports en and en-US locales. A complex Consumer would support multiple locales.	Complex	Complex
User Categories	Producers may be able to personalize portlet markup/behavior based on user categorization. Producers and Consumer typically agree on (out-of-band) the semantics of user categories.	Complex	Complex

8 Practical Considerations

8.1 Fault Handling

WSRP specifies a number of faults that a Producer may return in response to various requests from a Consumer. If you are setting up a Consumer to aggregate portlets from a Producer, knowledge of the implication of these faults would help debug any problems. Note that Producers may return additional faults as dictated by underlying web service stack.

Each WSRP fault has an associated faultcode. Here is an example message from a Producer that requires registration in response to a request from a Consumer without the `registrationContext`.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <soapenv:Fault xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
      <faultcode>urn:InvalidRegistration</faultcode>
      <faultstring>Registration is required.</faultstring>
      <detail>
        <urn:InvalidRegistration/>
      </detail>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

Message 37: Fault Response

The faultstring in this message explains that the Consumer must supply `registrationContext` with the request. Also, note the detail element. This element contains the name of the faultcode qualified by `oasis:names:tc:wsrp:v1:types namespace`.

8.1.1 AccessDenied Fault

A Producer may return this fault when the Producer is unable to provide access to a portlet for a given operation due to security policy reasons.

Action: Since such security policy restrictions are implementation specific, consult the entity hosting the Producer to determine an appropriate action.

8.1.2 InconsistentParameters Fault

A Producer may return the `InconsistentParameters` fault when a Consumer supplies inconsistent data. For example, a Producer returns this fault when a Consumer tries to access a portlet with a `portletHandle` that was created during a different `registrationContext` than the one the Consumer supplied.

Action: This fault may occur because of implementation errors in the Consumer. Make sure that the `portletHandle` included in the request corresponds to either a `portletHandle` returned in the service description response, or one returned during `performBlockingInteraction` or `clonePortlet` operations invoked using the same `registrationContext`. If not, use the correct `registrationContext` or reset the to one of those offered in the service description response.

8.1.3 InvalidRegistration Fault

This fault indicates that the Consumer did not supply a `registrationContext` when registration is required, or the supplied `registrationContext` is invalid.

Action: This fault may occur because of implementation errors in the Consumer. Check the Producer's service description to see if the Producer requires registration. If so, register the Consumer with the Producer and resend the request with a valid `registrationContext`. If you have already registered your Consumer with the Producer, make sure that the `registrationContext` is included in the request. Also make sure that the `registrationContext` is the same as the one returned by the Producer. If the Producer still returns the same fault, check the message to see if you need to reregister the Consumer as the Producer may have invalidated previous registration.

8.1.4 InvalidCookie Fault

HTTP cookies are transient and may become invalid. A Producer returns this fault when the Consumer supplied cookie is no longer valid.

Action: The fault may occur due to Producer imposed timeout of cookies. Invoke the `initCookie` operation again and then invoke the operation that caused this fault with the new cookie. Since cookies returned by the Producer may be related to the session IDs returned in an implementation-specific manner, resend any data stored that the Producer stores in the session. For example, if the Producer indicated that it stores URL templates and/or user context data in sessions, resend URL templates and user context data with the request.

8.1.5 InvalidHandle Fault

A Producer returns this fault when the Consumer supplied `portletHandle` or `registrationHandle` is invalid.

Action: This fault may occur because of implementation errors in the Consumer. Make sure that the `portletHandle` included in the request corresponds to either a `portletHandle` returned in the Service Description response, or one returned by either `performBlockingInteraction` or `clonePortlet` operations invoked using the same `registrationContext`.

8.1.6 InvalidSession Fault

A Producer returns this fault when a Consumer-supplied session ID is invalid.

Action: This fault may occur due to Producer imposed timeout of sessions. Invoke the same operation without the `sessionID` and with any data that the Producer has indicated that it stores in the session. For example, if the Producer stores URL templates and/or user context data in sessions, resend URL templates and user context data with the request.

8.1.7 InvalidUserCategory Fault

A Producer returns this fault when a Consumer supplied `userCategory` name is not valid. Note that Consumer should not supply user categories not recognized by Producers.

Action: This fault may occur as a result of implementation errors in the Consumer. Make sure that the `userCategory` is one of those supplied by the Producer in its service description response.

8.1.8 MissingParameters Fault

A Producer returns this fault when some required data is missing in the request.

Action: This fault may occur because of implementation errors in the Consumer. Since the set of parameters required are specific to each operation, refer to the description of the operation and make sure that all the required parameters are included in the request.

8.1.9 OperationFailed Fault

This is the most generic fault that a Producer can return for any request from a Consumer. In general, well-behaving Producers return this fault only when the condition that caused the fault cannot be described with any other WSRP fault.

Action: If you encounter this fault, look into the faultstring for any clues, and if no meaningful explanation is found, contact the entity hosting the Producer for help.

8.1.10 PortletStateChangeRequired Fault

A Producer throws this fault when the portlet needs to update its persistent state during a `performBlockingInteraction` but the Producer cannot allow it since the Consumer is not ready for state changes.

During `performBlockingInteraction`, the `portletStateChange` flag in the request dictates persistent state changes. When the value of this flag is set to `readWrite` or `cloneBeforeWrite`, the producer can allow the portlet to make persistent state changes. A persistent state change may cause the Producer to return a new `portletContext`. In case the Consumer is not capable of accepting the new `portletContext`, it may instead set the value of the `portletStateChange` flag to `readOnly`, which would cause this fault.

Action: If possible, configure the Consumer to allow state changes. Note that portlets may not function correctly when prevented from making persistent state changes. If you find that the portlet is not usable without persistent state changes and if you are unable to configure the Consumer to allow state changes, consider not using the portlet.

8.1.11 UnsupportedLocale Fault

A Producer may throw this fault when a portlet is unable to generate markup in the requested locale.

Action: This fault may occur because of implementation errors in the Consumer. Since the portlet is not capable of generating markup in the requested locale, restrict the Consumer to supply one of the locales indicated as supported by the portlet’s metadata.

8.1.12 UnsupportedMimeType Fault

This is similar to the `UnsupportedLocale` fault, and a Producer throws this fault when a portlet cannot generate markup in the requested MIME type.

Action: This fault may occur because of implementation errors in the Consumer. Since the portlet is not capable of generating markup in the requested MIME type, restrict the Consumer to supply one of the MIME types indicated as supported by the portlet’s metadata.

8.1.13 UnsupportedMode Fault

A Producer throws this fault when it cannot invoke a portlet in a given mode.

Action: This fault may occur because of implementation errors in the Consumer. Restrict the Consumer to supply a mode indicated as supported by the portlet’s metadata.

8.1.14 UnsupportedWindowState Fault

A Producer throws this fault when a portlet cannot be invoked in a given window state.

Action: This fault may occur because of implementation errors in the Consumer. Restrict the Consumer to supply a window state supported by the portlet

8.2 Localization

Most of the responses from a Producer include string values, such as names, descriptions, hints, etc. For example, the description of a portlet includes description, title, short-title, keywords etc expressed each with a string value and a locale. By default, Producers often use a fixed locale (usually the Producer’s default locale) for these values. In order to let Consumers obtain such values in some other locale (or more than one locale), WSRP allows Consumers to supply a list of preferred locales. Here are the operations that allow Consumers to pass locales:

1. `getServiceDescription`: Consumer can send a desiredLocales array with a `getServiceDescription` request. If supported, the Producer will return all strings in the given locales.
2. `getPortletDescription`: Consumer can send a desiredLocales array with a `getPortletDescription` request. If supported, the Producer will return all strings in the given locales.
3. `getPortletPropertyDescription`: Consumer can send a desiredLocales array with a `getPortletPropertyDescription` request. If supported, the Producer will describe the portlet’s properties in the given locales.
4. `getMarkup`: With a `getMarkup` request, the Consumer can send an array of locales to request the Producer to return the markup and the preferredTitle in one of the given

locales.

In all these cases, the Producer has the following options to send a localized string value:

1. Return the value as a `LocalizedString`, with a string value and a locale. For example, in Message 2, P Inc returned the description and title with a string value and an `xml:lang` attribute for the locale.
2. Return the value as a `LocalizedString` with a string value, an `xml:lang` attribute for the locale of the string value, and a resourceName attribute. The purpose of the resourceName attribute is to return string values in multiple locales with `getServiceDescriptionResponse`, `getPortletDescriptionResponse` and `getPortletPropertyDescriptionResponse` messages. Each of the messages can include a `resourceList` array. The `resourceList` array can contain several `Resource` elements. Each `Resource` element includes the resourceName, and a `ResourceValue` with string values expressed in the locale indicated by an `xml:lang` attribute.

To illustrate how a Producer can localize its response with resources, let us revisit Scenario 1, and consider that C Inc requested for strings in "en" and "de" locales. To such a request, P Inc may return the following response.

```
<urn:getServiceDescriptionResponse
xmlns:urn="urn:oasis:names:tc:wsrp:v1:types">
  <urn:requiresRegistration>false</urn:requiresRegistration>
  <urn:offeredPortlets>
    <urn:portletHandle>portfolioManager</urn:portletHandle>
    <urn:markupTypes>
      <urn:mimeType>text/html</urn:mimeType>
      <urn:modes>wsrp:view</urn:modes>
      <urn:windowStates>wsrp:normal</urn:windowStates>
      <urn:windowStates>wsrp:minimized</urn:windowStates>
      <urn:windowStates>wsrp:maximized</urn:windowStates>
      <urn:locales>en</urn:locales>
      <urn:locales>de</urn:locales>
    </urn:markupTypes>
    <urn:description resourceName="_description" xml:lang="en">
      <urn:value>Manages portfolios</urn:value>
    </urn:description>
    <urn:title resourceName="_title" xml:lang="en">
      <urn:value>Manage Your Portfolios</urn:value>
    </urn:title>
  </urn:offeredPortlets>
  <urn:locales>en</urn:locales>
  <urn:locales>de</urn:locales>
  <urn:resourceList>
    <urn:resources resourceName="_description">
      <urn:values xml:lang="de">
        <urn:value>Verwaltet die Portfolios</urn:value>
      </urn:values>
    </urn:resources>
    <urn:resources resourceName="_title">
      lt;urn:values xml:lang="de">
        lt;urn:value> Ihr Portfolio verwalten</urn:value>
      </urn:values>
    </urn:resources>
  </urn:resourceList>
</urn:getServiceDescriptionResponse>
```

Message 38: Get Service Description Response with Resources

This message is similar to Message 2, except for the parts shown in bold. In this response, both the description and title include resourceName attributes and corresponding resources. For each resourceName, the response includes a resources element with the resource value in "de" locale.

8.3 Extensions

Although the WSRP 1.0 Specification accounts for the most common scenarios, implementers may find a need to extend WSRP to deal with more advanced scenarios or scenarios not addressed by the specification. To account for such scenarios, WSRP Specification allows implementations to extend most of the WSRP data structures to include additional data as extensions.

Extensions are implementation-specific. A Producer and Consumer must agree to the purpose and semantics of the extension before extending any WSRP data structure. Note that Producers using extensions are not guaranteed to function correctly if a Consumer does not supply extended data required by the Producer. If you are implementing a Producer, consider making any extensions optional so that the Producer can interoperate with Consumers that do not supply extensions.

The WSRP specification requires that Producers and Consumers declare extensions in namespaces other than the one used by the WSRP specification to avoid conflicts with future versions of the WSRP specification.

If you find that WSRP specification does not represent some specific use case and you are relying on extensions to address the use case, please email your use case to wsrp-comment@lists.oasis-open.org.

8.4 Form Parameters and Multipart Upload

When a Consumer sends a `performBlockingInteraction` request, Producers reconstruct the input from the supplied InteractionParams. The InteractionParams can include form parameters as well as uploaded data. For example, when a HTML markup includes a form with an enctype attribute with value `multipart/form-data`, and one or more input controls of type `file`, the Consumer must extract the uploaded data from the incoming HTTP request, and send the same to the Producer. The Consumer has the following options of passing such data via InteractionParams to the Producer:

1. The Consumer can send the entire multipart request (including all uploaded files and other form parameters) as a single `uploadContexts` element, or one `uploadContexts` element per part in the incoming multiple request.
2. If the form also includes input controls of types other than file, the Consumer can send each value of the control as a `formParameters` element, or as an `uploadContexts` element.

The `uploadContexts` element includes the uploaded data (as `uploadData`), its MIME type (as `mimeType`), and any MIME headers sent by the browser to the Consumer (as `mimeAttributes`). Since browsers may send a variety of headers while uploading files, we recommend Consumer implementations to include all those attributes to let the Producer reconstruct the request.

9 References

1. Web Services for Remote Portlets 1.0 Specification, http://www.oasis-open.org/committees/download.php/3343/oasis-200304-wsrp-specification-1.0.pdf
2. Web Services for Remote Portlets 1.0 White Paper, http://www.oasis-open.org/committees /download.php /2634/WSRP Whitepaper - rev 2.doc

3. WSDL for WSRP Interfaces, http://www.oasis-open.org/committees/wsrp/specifications/version1/wsrp_v1_interfaces.wsdl
4. WSRP data types, http://www.oasis-open.org/committees/wsrp/specifications/version1/ wsrp_v1_types.xsd
5. WSRP WSDL Report, http://www.oasis-open.org/committees/wsrp/presentations/012003 / wsrp-wsdl-report.doc
6. WSRP Conformance Test Kit, http://www.alphaworks.ibm.com/tech/wsrptk
7. WSRP Conformance Statements, http://www.oasis-open.org/committees/download.php/6018/WSRP Conformance Statements.xls
8. RFC 2965, HTTP State Management Mechanism, http://www.ietf.org/rfc/rfc2965.txt
9. WSRP Use Profiles, http://www.oasis-open.org/committees/download.php/3073/WSRP Use Profiles.doc

Appendix: Acknowledgements

The editors/contributors of this Primer would like to acknowledge the ideas, material, and feedback provided by Polina Alber, Olin Atkinson, Christopher Coco, William Cox, Howard Crow, Michael Freedman, Ricky Frost, Simon Godik, Scott Goldstein, Lars Hofhansl, Richard Jacob, Andre Kramer, Avi Klein, Jon Klein, Carsten Leue, Khurram Mahmood, Farrukh Najmi, Fubini Ross, Yossi Tamari, and Rich Thompson,

Appendix: Notices