AWSP
Asynchronous Web Services Protocol

Keith Swenson, MS2 Inc.

Jeffrey Ricker, Trans-enterprise Integration Corp.

Draft, 05-April-2002

This document represents a proposal for a protocol standard. This version has not been approved for adoption. Future version may change substantially from this version.

Abstract

A standard protocol is needed to integrate asynchronous services across the Internet or intranet and provide for their interaction. The integration and interactions consist of control and monitoring of the services. Control means creating the service, setting up the service, starting the service, stopping the service, being informed of exceptions, being informed of the completion of the service and getting the results of the service. Monitoring means checking on the current status of the service and getting a history of the execution of the service. The protocol should be lightweight and easy to implement, so that a variety of devices and situations can be covered.

The Asynchronous Web Services Protocol (AWSP) is a proposed way to solve this problem through use of Simple Object Access Protocol (SOAP), and by transferring structured information encoded in XML. A new set of SOAP methods are defined, as well as the information to be supplied and the information returned in XML that accomplish the control of generic asynchronous services. This protocol has applicability to business-to-business e-commerce because business processes are an asynchronous service, and they also have the need to be able to call asynchronous services outside of themselves.

This document will:

- Provide an executive overview
- Specify the goals of AWSP

- Explain how resource (object) model works and how URIs are used to invoke methods of those resources.
- Explain how to encode data to be sent or received.
- Specify preliminary details of the interface methods and parameters

Table of contents

1.0 Executive Overview

1.1 Summary

This protocol offers a way to start an instance of an asynchronous web service (AWS), monitor it, control it, and be notified when it is complete. This service instance can perform just about anything for any purpose. The key aspect is that the service instance is something that one would like to start remotely, and it will take a long time to run to completion. Short-lived services would be invoked synchronously with SOAP and one would simply wait for completion, but because these process instances last anywhere from a few minutes to a few months, they must be invoked asynchronously.

What does this have to do with electronic commerce? While this protocol is useful for starting and controlling service instances of any type, the need for this capability is of high demand in business-to-business e-commerce. A business process is a service that may run for days or even months, so this protocol will be useful for starting business processes. It is also the case that business processes provide sequencing of other services, and thereby need a way to start an external generic asynchronous service, and to be told when it completes. So this protocol is immensely useful to the business-to-business e-commerce community, but has little actual specific to e-commerce.

How does it work? You must start with the URI of a service definition. An SOAP request to this URI will cause this service definition to generate a service instance, and return the URI of this new service instance that is used for all the rest of the requests. The service instance can be provided with data (any XML data structure) by another SOAP request. The current state of the service instance can be retrieved with another SOAP request. The

service instance can be paused or resumed with another SOAP request. There is also a pair of requests that may be used to give input data to the service instance, and to ask for the current value of the output data.

When it is done? The service instance runs asynchronously and takes whatever time it needs to complete. The originating program can, if it chooses, keep polling the state of the service instance in order to find out when it is complete. This will consume resources unnecessarily both on the originating side as well as the performing side. Instead, the originator may provide the service instance with the URI of an observer.

When the service instance is completed it will do a SOAP request back to this URI in order to notify the originator. This allows the originator to be put to sleep, freeing up operating system as well as network resources while waiting for the service instance to complete.

1.2 Not-so-technical executive summary

What does this mean in English? Most existing Internet protocols like HTTP are based on an unwritten assumption of instant gratification. If a client asks for any resource that takes longer than about a minute to generate, then the request times out, that is, it fails. We call anything on the Internet like HTML pages and GIF images a *resource*. Most resources such as web pages are static or require a very simple database query to create, so they easily meet the instant gratification requirement.

As we have applied Internet technology to more and more scenarios, this assumption of instant gratification has become more strained. A good example is wireless Internet. With wireless, the resource may take more than a minute to generate simply because of a poor connection.

A more telling example is electronic commerce. In commerce, it may not be a simple database query that generates a document but rather an entire corporate business process with a human approval involved. Very few corporate business processes, especial those requiring management approval, take less than a minute to complete.

What is needed is a simple ability to ask for a resource and for that resource to be able to respond, “The information isn’t ready yet. Where would you like me to send it when I’m done?” That is what we mean by *start an instance of a generic asynchronous service and be notified when it is complete*. Someone asking for the resource should be able to pester, just like in the real world, with questions like, “Are you done yet? Where is that document I asked for?” That is what we mean by *monitor*. Finally, we should be able to change our mind in mind process, just like in the real world, with statements like, “Change that to five widgets, not six.” That is what we mean by *control*.

With such a protocol, we should be able to integrate not just applications but business processes, which is what electronic commerce is really all about. With such a protocol we should also be able to integrate within and between enterprises much faster (hence the name trans-enterprise integration) because suddenly we are able to have manual processes look and feel to everything else on the Internet as if it were actually automated.

Here is an example. An AWSP message is sent to a server requesting inventory levels of a certain part number. The server responds to the requestor “The information isn’t ready yet. Where would you like me to send it when I’m done?” The server then sends a message to Steve’s two-way pager in the warehouse asking him to type in the inventory level of the certain part number. After a coffee break, Steve duly types in the number. The

server creates the proper message and responds to the requestor. To the outside world, an electronic message was sent and an electronic message was received. The result is automated inventory level tracking. Nobody need know that Steve walked down the aisle and counted.

1.3 Discussion of this Draft

Discussions on this draft are carried out on a mailing list:

awsp-wg@awsp.info

Comments on the draft should be sent to that email address. Please check the following web site for instructions on how to join the mailing list and for information on other documents from the same working group:

http://www.awsp.info

2 Goals

2.1 Problem Statement

Not all services are instantaneous.

A standard protocol is needed to integrate asynchronous services (processes or work providers) across the Internet/intranet and provide for their interaction. The integration and interactions consist of control and monitoring of the service. Control means creating the service, setting up the service, starting the service, stopping the service, being informed of exceptions, being informed of the completion of the service and getting the results of the service. Monitoring means checking on the current status of the service and getting a history of the execution of the service.

The protocol should be lightweight and easy to implement, so that a variety of devices and situations can be covered.

2.2 Things to Achieve

In order to have a realizable agreement on useful capabilities in a short amount of time, it is important to be very clear about the goals of this effort.

- The protocol should not reinvent anything unnecessarily. If a suitable standard exists, it should be used rather than re-implemented a different way.
- The protocol should be consistent with XML Protocol and SOAP.
- The protocol should be the minimal necessary to support a generic asynchronous service. This means being able to start, monitor, exchange data with, and control a generic asynchronous service on a different system.
- The protocol must be extensible. The first version will define a very minimal set of functionality. Yet a system must be able to extend the capability to fit the needs of a particular system, such that high level functionality can be communicated which gracefully degrades to interoperate with systems that do not handle those extensions.

- Like other Internet protocols, AWSP should not require or make any assumptions about the platform or the technology used to implement the generic asynchronous service.

- Terseness of expression is not a goal of this protocol. Ease of generation and parsability should be favored over compactness.

Regarding human readability, the messages should be self-describing for the programmer, but they are not intended for direct display for the novice end user. This specification attempts to adhere to Eric S. Raymond's ninth principle: "Smart data structures and dumb code works a lot better than the other way around," or, paraphrased from Frederick P. Brooks, "Show me your [code] and conceal your [data structures], and I shall continue to be mystified. Show me your [data structures], and I won't usually need your [code]; it'll be obvious." [15]

2.3 Things not part of the goals

It is also good practice to clearly demark those things that are not to be covered by the first generation of this effort:

- The goals of AWSP do not include a way to set up or to program the generic services in any way. Especially for the case where the service is a workflow service, AWSP does not provide a way to retrieve or submit process definitions. The service can be considered to be a "black box" which has been pre-configured to do a particular process. AWSP does not provide a way to discover what it is that the service is really doing, only that it does it (given some data to start with) and some time later completes (providing some result data back).

- AWSP will not include the ability to perform maintenance of the asynchronous web service such as installation or configuration.

- AWSP will not support statistics or diagnostics of collections of asynchronous web services. AWSP is designed for the control and monitoring of individual asynchronous web services.

- AWSP is not designed to handle the transfer of large amounts of process relevant data. The data associated with, and stored within, a service instance is not anticipated to be more than 64K Bytes, and in most cases will be quite a bit less than this. If larger amounts of data than this are needed then it is anticipated that this data would be placed into some form of external service and that the AWSP will only need to carry the URI to that data. For example, a document would be placed on a web server and given the URI would be retrieved through normal HTTP.

- AWSP does not specify security. Rather, it relies on transport or session layer security. AWSP can adopt SOAP –specific security protocols once they are finalized.

These may be added in a later revision, but there is no requirement to support these from the first version, and so any discussion on these issues should not be part of AWSP working group meetings.

2.4 Audience

This document is intended for vendor organizations who wish to implement the AWSP protocol in e-commerce process flow engines, and also for consultants, VARs, or third party developers who wish to make a AWSP wrapper for an existing system service in order to integrate that service into a process flow or workflow environment.

2.5 Terminology

Web Service: a program or service accessible using XML and Internet technologies. Such a service might be found using UDDI, and might be described using WSDL. The typical interaction is: an originator makes a request is made to a web service, which performs the operation, and returns the result.

Asynchronous Web Service: A web service or set of web services designed around a mode of operation where a request is made to start an operation, and a later separate request is made to communicate the results of the operation. A number of requests may be made in between in order to control and monitor the asynchronous operation. The results of the operation may be delivered either by polling requests from the originator, or else by a notification request originated by the performer.

Method: An individual interoperable function is termed a “method”. Each method may be passed a set of request parameters and return a set of response parameters.

Resource types: Methods are divided into different groups to better identify their context. The primary groups of methods required for interoperability are named Instance, Factory, and Observer.

Instance Resource: This is the resource implemented by the Web Service that is actually performing the requested work. These resources allow for the actual monitoring and controlling of the work.

Factory Resource: This is the resource implemented by the service instance factory. Methods are provided to start new service instances, to list or search for existing instances, and to provide definitional information about the instances.

Observer Resource: This is a resource that a web service must implement in order to receive notification events from the service instance.

Context Data: The XML data sent to initiate the service.

Result Data: The XML data created by the successful completion of the service.

2.6 Related Documents

An understanding of SOAP and how it works is assumed in order to understand this document.

2.7 Notation conventions

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

The following namespace prefixes are used throughout this document:

Prefix	Namespace URI	Definition
aws	http://www.awsp.info/spec/1.0/	AWSP namespace
env	http://schemas.xmlsoap.org/soap/envelope/	Envelope namespace from SOAP 1.1
enc	http://schemas.xmlsoap.org/soap/encoding/	Encoding namespace from SOAP 1.1
xsd	http://www.w3.org/2001/XMLSchema	XML Schema namespace

Table 1 Namespaces

This specification uses an informal syntax we call *pseudo-XML* to describe the XML grammar of an AWSP document. This syntax is similar to that employed by the WSDL 1.1 specification

Convention	Example
The syntax appears as an XML instance, but the values indicate the data types instead of values.	`<p:tag name="nmtoken"/>`

Paragraphs within tags are the description of the tag and should be thought of as commented out with <!-- -->	`<p:tag>`
	`  longer description of the`
	`  purpose of the tag.`
	`</p:tag>`

Characters are appended to elements and attributes as follows: "?" (0 or 1), "*" (0 or more), "+" (1 or more).	`<p:tag>*`
Elements names ending in "…" indicate that elements/attributes irrelevant to the context are being omitted or they are exactly as defined previously.	`<p:tag.../>`
Grammar in bold has not been introduced earlier in the document, or is of particular interest in an example.	`<p:tag/>`
“Extensible element” is a placeholder for elements from some "other" namespace (like ##other in XSD).	`<-- extensible element -->`
The XML namespace prefixes (defined above) are used to indicate the namespace of the element being defined	
Examples starting with <?pseudo-xml?> contain enough information to conform to this specification; others examples are fragments and require additional information to be specified in order to conform.	`<?pseudo-xml?>`

Table 2 Pseudo-XML documentation conventions

Formal syntax is available in supplementary XML Schema and WSDL specifications in the document.

3 Resource Model

3.1 Resources Type Overview

For the support of an asynchronous web service (AWS), three types of web services are defined to match the three roles of the interaction: Instance, Factory, and Observer. A

Web Service type is distinguished by the group of operations it supports, and so there are three groups of operations.

For the purpose of discussion we describe the different role as if they came from different web services, but there is no requirement that each resource type be provided by a different web service; it is possible that a single web service may play more than one role in the interaction simply by implementing all the methods in more than one resource type.

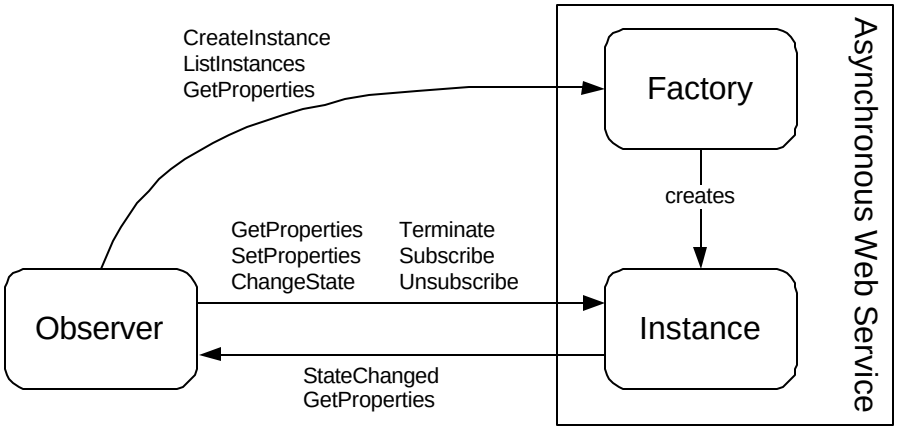

Figure 1 Resource types of an asynchronous web service and the methods they use

3.1.1 Instance

The Instance resource is the actual "performance of work". It embodies the context information that distinguishes one performance of an AWS from another. Every time the service is to be invoked, a new process instance is created and given its own resource identifier. A process instance resource can be used only once: it is created, then it can be started, it can be paused, resumed, terminated. If things go normally, it will eventually complete.

When a service is to be enacted, a requestor will reference a service factory's resource identifier and create an instance of that service. Since a new instance will be created for each enactment, the service factory may be invoked (or instantiated) any number of times simultaneously. However, each service instance will be unique and exist only once. Once created, a service instance may be started and will eventually be completed or terminated.

For example, a company may need to, at certain times, run a CICS transaction to accomplish some financial function. This could be started in a proprietary or platform specific way, but the right client libraries and version would have to be installed at every location that needed to be able to trigger the transaction. Alternately the service could be enabled for AWSP with a relatively simple CGI wrapper that implemented the service instance interface that would allow the transaction to be triggered by any tool that supported AWSP.

3.1.2 Factory

The Factory resource represents a "way of doing some work". It is the most fundamental resource required for the interaction of generic services. It represents the description of a service's most basic functions, and is the resource from which instances of a service will be created. Since every service to be enacted must be uniquely identifiable by an

interoperating service or service requestor, the factory will provide a resource identifier. When a service is to be enacted, this resource identifier will be used to reference the desired process to be executed. A service might be a set of tasks carried out by a group of individuals, or it might be set of machine instructions that make up a executable program, or it might be any combination of these. The important point to remember about a service factory is that while it embodies the knowledge of how a work it performed, it does not actually do the work.

3.1.3 Observer

The Observer resource provides a means by which a service instance may communicate information about events occurring during its execution, such as its completion or termination. Third-party resources may have an interest in the status of a given service instance for various organizational reasons. The Observer group will provide this information by giving a service instance the resource identifier of the requestor, which will be the observer of that service instance.

3.2 URI

Each resource has an URI address, called the Key. A given implementation has complete control over how it wishes to create the URI that names the resource. It must stick to a single method of producing these URI Keys, so that the names can serve as a unique id for the resource involved. The receiving program should treat it as an opaque value and not assume anything about the format of the URI.

3.3 Parameters

Associated with a method will be any number of named values that form the parameters of the method. These will be represented in the body of the SOAP request message. Some parameters are optional and may be omitted. In general, extra parameters can be added for extensibility; implementations should ignore any parameters that it does not understand.

3.4 Returned Values

The result of request is a block of data encoded in XML in the body of the SOAP message. Each method defines the values that it will return. Some values are marked as optional and are not required, the rest are required to be part of the return set. An implementation of a method may return more than the required set of named value, including new values unique to that implementation, as long as ignoring those values has no effect on the usefulness of the required values. Clients of such implementation should be coded to properly handle responses that do not include those extended values.

4 Protocol

4.1 SOAP

Simple Object Access Protocol (SOAP) [8] is a protocol that defines a simple way for two programs to exchange information. The protocol consists of a client program that initiates a request to a server program. Any given program may be capable of being both a client and a server; our use of these terms refers only to the role being performed by the

program for a particular connection, rather than to the program's capabilities in general. The request involves the sending of a request message from the client to the server. The response involves the sending of a response message from the server back to the client. Both the request and response messages conform to the SOAP message format.

Certain message may carry entities that can be understood as the "information payload" of the message. AWSP extends SOAP by specifying specific methods and entities.

4.1.1 Envelope

The root tag of an AWSP message is a SOAP envelope as defined by the SOAP standard.

4.1.2 Header

The message must contain a SOAP header as per the SOAP standard, used for all the regular purpose such as addressing and routing the message. AWSP does not make any changes here.

4.1.3 Body

In accordance with SOAP, there must be a body tag within the envelope tag. The AWSP protocol requires that two things appear in the body of the SOAP request.

First, it must contain either a Request element or a Response element. These are elements which are the same for all operations, and carry pieces of information common across all operations.

Second, AWSP requires that there be one of the following elements within the body which represents the information needed for a specific operation:

- GetProperties.Request
- GetProperties.Response
- SetProperties.Request
- SetProperties.Response
- CreateInstance.Request
- CreateInstance.Response
- ListInstances.Request
- ListInstances.Response
- ChangeState.Request
- ChangeState.Response
- StateChanged.Request
- StateChanged.Response
- Notify.Request
- Notify.Response
- Subscribe.Request
- Subscribe.Response
- Unsubscribe.Request
- Unsubscribe.Response
- env:Fault

These tags and their contents are described in detail in the sections on the specific operations.

4.2 Request

SenderKey: The request MAY specify the URI or key of the resource that originated the request. This may be redundant with similar specifier in the transport layer.

ReceiverKey The request MUST specify the key of the resource that the request is being made to. This may be redundant with similar specifier in the transport layer.

ResponseRequired: This optional tag may contain the following values: Yes, No, or IfError. If the value specified is Yes, a response must be returned for this request in all cases, and it must be processed by the requesting resource. If the value specified is No, a response may, but need not be returned for this request, and if one is returned it may be ignored by the requesting resource. If the value specified is IfError, a response only needs to be returned for this request in the case where an error has occurred processing it, and the requesting resource must process the response. If this tag is not specified, the default value is assumed to be Yes.

RequestID: The requester may optionally specify a unique ID for the request. If present, then this ID must be returned to the requester in the RequestID tag of the response in order to correlate that response with the original request. The value is assumed to be an opaque value.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request xmlns:aws="http://www.awsp.info/spec/1.0/">
      <aws:SenderKey>? The URI of the sender </aws:SenderKey>
      <aws:ReceiverKey> The URI of the receiver </aws:ReceiverKey>
      <aws:ResponseRequired>Yes|No|IfError</aws:ResponseRequired>
      <aws:RequestID>?
        Unique ID for message correlation by the requestor
      </aws:RequestID>
    </aws:Request>
  </env:Header>
  <env:Body>
    ...
  </env:Body>
</env:Envelope>
```

Example 1 Request header

```
<xsd:element name="Request">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="SenderKey" type="xsd:anyURI" minOccurs=”0”/>
      <xsd:element name="ReceiverKey" type="xsd:anyURI"/>
      <xsd:element name="ResponseRequired" type=”YesNoIfError”>
      <xsd:element name="RequestID" type="xsd:anyURI" minOccurs=”0”/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:simpleType name=”YesNoIfError”>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Yes"/>
    <xsd:enumeration value="No"/>
    <xsd:enumeration value="IfError"/>
  </xsd:restriction>
</xsd:simpleType>
```

Schema 1 Request header

4.3 Response

The presence of a Response tag in the body indicates that this is an answer to a request.

`SenderKey`: The request MUST specify the URI or key of the resource that originated the response. This may be redundant with similar specifier in the transport layer.

`ReceiverKey` The request MAY specify the key of the resource that the response is being made to. This may be redundant with similar specifier in the transport layer.

Note that the ReceiverKey is mandatory in a request and the SenderKey is mandatory in a response. The purpose is to enforce keys upon AWSP resources without placing an unnecessary burden on resources that are merely employing AWSP resources. For instance, a Java program that instantiates an AWS may not know its own URL.

`RequestID`. If the original request had a RequestID tag, then the response must carry one with that value in it. The requester can use this ID to correlate the response with the original request.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response xmlns:aws="http://www.awsp.info/spec/1.0/">
      <aws:SenderKey> The URI of the sender </aws:SenderKey>
      <aws:ReceiverKey>? The URI of the receiver </aws:ReceiverKey>
      <aws:RequestID>?
        Unique ID for message correlation by the requestor
      </aws:RequestID>
    </aws:Response>
  </env:Header>
  <env:Body>
    ...
  </env:Body>
</env:Envelope>
```

Example 2 Response header

```
<xsd:element name="Response">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="SenderKey" minOccurs=”0”/>
      <xsd:element ref="ReceiverKey"/>
      <xsd:element ref="RequestID" minOccurs=”0”/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Schema 2 Response header

5 Resource Methods

5.1 Instance Resource

All resources that represent the execution of a long-term asynchronous service must implement the Service Instance resource. The purpose of this resource type is to allow

the work to proceed asynchronously from the caller. The Instance represents a unit of work, and a new instance of the Instance resource must be created for every time the work is to be performed.

The performing of the work may take anywhere from minutes to months, so there are a number of operations that may be called while the work is going on. While the work is proceeding, AWSP requests can be used to check on the state of the work. If the input data has changed in the meantime, new input values may be supplied to the Instance, though how it responds to new data is determined by details about the actual task it is performing. Early values of the result data may be requested, which may or may not be complete depending upon the details of the task being performed. The results are not final until the unit of work is completed. When the state of the Instance changes, it can send events to the Observer informing it of these changes. The only event that is absolutely required is the "completed" or "terminated" events that tell the requesting resource that the results are final and the Instance resource may be disappearing.

While a business process will implement Instance, it is important to note that there are also many other types of resources that will implement the Instance resource; it will also be implemented on any discrete task that needs to be performed asynchronously. Thus a wrapper for a legacy CICS transaction would implement the Instance resource so that that legacy application could be called and controlled by any program that speaks AWSP. A driver for an actual physical device, such as a numerical milling machine, would implement the Instance resource if that device were to be controlled by AWSP. Any program to be triggered by a process flow system that takes a long time to perform should implement the Instance resource, for example a program that automatically backs up all the hard drives for a computer. Since these resources represent discrete units of work (which have no subunits represented within the system) these resources will not need to have any Activities.

5.1.1 Instance Resource Properties

Key: A URI that uniquely identifies this resource.

PortTypes: a list of the resource types that this resource supports.

State: The current status of this resource. Please see the details on the status property later in section on “Status Property”. This property is not directly settable, but can be affected indirectly through the ChangeState command.

Name: A human readable identifier of the resource. This name may be nothing more than a number.

Subject: A short description of this process instance. This property can be set using SetProperties.

Description: A longer description of this process instance resource. This property can be set using SetProperties.

ValidStates: A list of state values allowed by this resource. This is the list of states to which the current instance can transition.

FactoryKey: URI of the factory resource from which this instance was created.

Observers: A collection of URIs of registered observers of this process instance, if any exist.

ContextData: Context-specific data that represents the values that the service execution is expected to operate on.

ResultData: Context-specific data that represents the current values resulting from process execution. This information will be encoded as described in the section Process Context and Result Data above. If result data are not yet available, the ResultData element is returned empty.

Priority: An indication of the relative importance of this instance. This value will be an integer ranging from 1 to 5, 1 being the highest priority. The default value is 3.

LastModified: The date of the last modification of this instance, if available.

History**:** See the definition of the history type later in this document.

```
<?pseudo-xml?>
...
<aws:Key> URI </aws:Key>
<aws:PortType>*Instance</aws:PortType>
<aws:State>open.notrunning</aws:State>
<aws:Name> string </aws:Name>
<aws:Subject> string </aws:Subject>
<aws:Description> string </aws:Description>
<aws:RequestingUser> user ID </aws:RequestingUser>
<aws:AccessUsers>
  <aws:AccessUser>* user ID </aws:AccessUser>
</aws:AccessUsers>
<aws:ValidStates>open.running etc.</aws:ValidStates>
<aws:FactoryKey> URI </aws:FactoryKey>
<aws:Observers>
  <aws:ObserverKey>* URI </aws:ObserverKey>
</aws:Observers>
<aws:ContextData>
  <-- extensible element -->
</aws:ContextData>
<aws:ResultData>
  <-- extensible element -->
</aws:ResultData>
<aws:Priority>3</aws:Priority>
<aws:LastModified> date-time group </aws:LastModified>
<aws:History xlink:href=”url”/>
...
```

Example 3 Instance resource properties

```
<xsd:complexType name=”instanceProperties”>
  <xsd:sequence>
    <xsd:element name="Key" type="xsd:anyURI"/>
    <xsd:element name="PortType" type="PortType" maxOccurs="unbounded"/>
    <xsd:element name="State" type="stateType"/>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="Subject" type="xsd:string"/>
    <xsd:element name="Description" type="xsd:string"/>
    <xsd:element name="ReqestingUser" type="userType"/>
    <xsd:element name="AccessUsers">
```

```
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name=”AccessUser” type="userType" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="ValidStates" type="stateType" maxOccurs="unbounded"/>
   <xsd:element name="FactoryKey" type="xsd:anyURI"/>
   <xsd:element name="Observers">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name=”ObserverKey” type="xsd:anyURI" maxOccurs="unbounded"/>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
   <xsd:element name="ContextData" type="xsd:anyType"/>
   <xsd:element name="ResultData" type="xsd:anyType"/>
   <xsd:element name="Priority">
     <xsd:restriction base="xsd:integer">
       <xsd:minInclusive value=”1”/>
       <xsd:maxInclusive value=”5”/>
     </xsd:restriction>
   </xsd:element>
   <xsd:element name="LastModified" type="xsd:date"/>
   <xsd:element name="History" type="historyType"/>
 </xsd:sequence>
</xsd:complexType>
```

Schema 3 Instance resource properties

5.1.2 GetProperties

This is a single method that returns all the values of all the properties of the resource.

`GetProperties.Request`: This is the main element present in the SOAP Body element.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:GetProperties.Request/>
  </env:Body>
</env:Envelope>
```

Example 4 Instance resource GetProperties method request

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response..>
  </env:Header>
  <env:Body>
    <aws:GetProperties.Response>
      <-- properties -->
    </aws:GetProperties.Response>
  </env:Body>
</env:Envelope>
```

Example 5 Instance resource GetProperties method response

```
<xsd:element name="GetProperties.Request"/>
<xsd:element name="GetProperties.Response" type=”instanceProperties”/>
```

Schema 4 Instance resource GetProperties method

5.1.3 SetProperties

All resources implement SetProperties and allow as parameters all of the settable properties. This method can be used to set at least the displayable name, the description, or the priority of a process flow resource. This is an abstract interface, and the resources that implement this interface may have other properties that can be set in this manner. All of the parameters are optional, but to have any effect at least one of them must be present. This returns the complete info for the resource, just as the GetProperties method does, which will include any updated values.

`Data`: A collection elements that represent the context of this Instance. The elements are from the schema defined by this resource. The context is considered to be the union of the previous context and these values, which means that a partial set of values can be used to update just those elements in the partial set having no effect on elements not present in the call.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:SetProperties.Request>
      <aws:Subject...>?
      <aws:Description...>?
      <aws:State...>?
      <aws:Priority...>?
      <aws:Data>
        <-- extensible element -->
      </aws:Data>
    </aws:SetProperties.Request>
  </env:Body>
</env:Envelope>
```

Example 6 Instance resource SetProperties method request

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response...>
  </env:Header>
  <env:Body>
    <aws:SetProperties.Response...>
      Returns the same response as GetProperties
    </aws:SetProperties.Response>
  </env:Body>
</env:Envelope>
```

Example 7 Instance resource SetProperties method response

```
<xsd:element name="SetProperties.Request">
  <xsd:complexType>
```

```
    <xsd:element ref=”Subject”/>
    <xsd:element ref=”Description”/>
    <xsd:element ref=”State”/>
    <xsd:element ref=”Priority”/>
    <xsd:element name=”Data” type=”xsd:anyType”>
  </xsd:complexType>
</xsd:element>
<xsd:element name="SetProperties.Response" type=”instanceProperties”/>
```

Schema 5 Instance resource SetProperties method

5.1.4 Terminate

This method cancels the execution of the service. It is used when the initiator is no longer interested in the service being performed When a Instance is terminated that is waiting on other Instances, then it should terminate those Instances that is waiting on.

Terminate returns all the property data of the service instance since this is likely to be the final interaction with the instance.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:Terminate.Request/>
  </env:Body>
</env:Envelope>
```

Example 8 Instance resource Terminate method request

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response..>
  </env:Header>
  <env:Body>
    <aws:Terminate.Response>
      <--properties-->
    </aws:Terminate.Response>
  </env:Body>
</env:Envelope>
```

Example 9 Instance resource Terminate method response

```
<xsd:element name="Terminate.Request"/>
<xsd:element name="Terminate.Response" type=”instanceProperties”/>
```

Schema 6 Instance resource Terminate method

5.1.5 Subscribe

To allow scalability, Instances will notify Observers when important events occur. Observers must register their URIs with the Instance in order to be

The subscribe method is a way for other implementations of the Observer Operation Group to register themselves to receive posts about changes in process instance state. Not all Instance resources will support this; those that do not will return an exception value that will explains the error.

`ObserverKey`: URI to a resource that both implements the Observer Operation Group and will receive the events

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
<env:Header>
  <aws:Request...>
  </env:Header>
<env:Body>
  <aws:Subscribe.Request>
    <aws:ObserverKey> URI </aws:ObserverKey>
    </aws:Subscribe.Request>
  </env:Body>
</env:Envelope>
```

Example 10 Instance resource Subscribe method request

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
<env:Header>
  <aws:Response...>
  </env:Header>
<env:Body>
  <aws:Subscribe.Response/>
  </env:Body>
</env:Envelope>
```

Example 11 Instance resource Subscribe method response

```
<xsd:element name="Subscribe.Request">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="ObserverKey"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="Subscribe.Response"/>
```

Schema 7 Instance resource Subscribe method

5.1.6 Unsubscribe

This is the opposite of the subscribe method. Resource removed from being observers will no longer get events from this resource. The URI of the resource to be removed from the observers list must match exactly to an URI already in the list. If it does match, then that URI will be removed. If it does not match exactly, then there will be no change to the service instance.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:Unsubscribe.Request>
      <aws:ObserverKey> URI </aws:ObserverKey>
    </aws:Unsubscribe.Request>
  </env:Body>
</env:Envelope>
```

Example 12 Instance resource Unsubscribe method request

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response...>
  </env:Header>
  <env:Body>
    <aws:Unsubscribe.Response/>
  </env:Body>
</env:Envelope>
```

Example 13 Instance resource Unsubscribe method response

```
<xsd:element name="Unsubscribe.Request">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="ObserverKey"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="Unsubscribe.Response"/>
```

Schema 8 Instance resource Unsubscribe method

5.1.7 ChangeState

5.2 Factory Resource

5.2.1 Factory Resource Properties

Key: A URI that uniquely identifies this resource. All resources must have a Key property.

PortType: a list of the resource types that this resource supports, in this case “Factory”.

Name: A human readable identifier of the resource. This name may be nothing more than a number.

Subject: A short description of this service. Note that the factory and the instance both have a subject. The subject of the factory should be general. The subject of an instance should be specific.

Description: A longer description of what the AWS will perform. . Note that the factory and the instance both have a subject. The subject of the factory should be general. The subject of an instance should be specific.

ValidStates: A list of state values allowed by instances of this service.

ContextDataSchema: An XML Schema representation of the context data that should be supplied when starting an instance of this process. This element either contains a link to a schema file or contains an xsd:schema element.

`ResultDataSchema`: An XML Schema representation of the data that will generated and returned as a result of the execution of this process. This element either contains a link to a schema file or contains an xsd:schema element.

`Expiration`**:** The minimum amount of time the service instance will remain accessible as a resource after it has been completed for any reason. The requester must plan to pick up all data within this time span of service completion. Data might remain longer than this, but there is no guarantee. The value is expressed as an XML Schema duration data type. For instance, 120 days is expresses as “P120D”.

```
<?pseudo-xml?>
...
<aws:Key> URI </aws:Key>
<aws:PortType>Factory</aws:PortType>
<aws:Name> xsd:string </aws:Name>
<aws:Subject> xsd:string </aws:Subject>
<aws:Description> xsd:string </aws:Description>
<aws:ValidStates>
  <aws:State>open.notrunning</aws:State>*
</aws:ValidStates>
<aws:ContextDataSchema href=”URL”/>
<aws:ResultDataSchema>
  <xsd:schema xmlns:xsd=”http://www.w3.org/2001/XMLSchema”>
    <-- schema specification -->
  </xsd:schema>
</aws:ResultDataSchema>
<aws:Expiration> xsd:duration </aws:Expiration>
...
```

Example 14 Factory resource properties

```
<xsd:complexType name=”factoryProperties”>
  <xsd:sequence>
    <xsd:element ref="Key"/>
    <xsd:element ref="PortType" maxOccurs="unbounded"/>
    <xsd:element ref="Name"/>
    <xsd:element ref="Subject"/>
    <xsd:element ref="Description"/>
    <xsd:element ref="ValidStates" maxOccurs="unbounded"/>
    <xsd:element name="ContextDataSchema" type=”schemaType”/>
    <xsd:element name="ResultDataSchema" type=”schemaType”/>
    <xsd:element name="Expiration" type="xsd:duration"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name=”schemaType”>
  <xsd:sequence>
    <xsd:
  </xsd:sequence>
  <xsd:attribute name=”href” type=”anyURI”/>
</xsd:complexType>
```

Schema 9 Factory resource properties

5.2.2 GetProperties

The Factory resource GetProperties method request is exactly the same as the Instance resource GetProperties request. The response returns the properties particular to the factory resource.

```
<?pseudo-xml?>
```

```
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:GetProperties.Request/>
  </env:Body>
</env:Envelope>
```

Example 15 Factory resource GetProperties method request

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response..>
  </env:Header>
  <env:Body>
    <aws:GetProperties.Response>
      <-- properties -->
    </aws:GetProperties.Response>
  </env:Body>
</env:Envelope>
```

Example 16 Factory resource GetProperties method response

5.2.3 CreateInstance

Given a process definition resource, this method is how instances of that process are created. There are two modes: create the process, with data, and start it immediately; or just create it and put the data on it and start it manually.

StartImmediately element holds a Boolean value to say whether the process instances that is created should be immediately started, or whether it should be put into an initial state for later starting by use of the “start” operation. If this tag is missing, the default value is “Yes”.

ObserverKey: holds the URI that will receive events from the created process instance. This observer resource (if it is specified) is to be notified of events impacting the execution of this process instance such as state changes, and most notably the completion of the instance.

Name: A human readable name of the new instance. There is no commitment that this name be used in any way other than to return this value as the name. There is no implied uniqueness constraints.

Subject: A short description of the purpose of the new instance.

Description: A longer description of the purpose of the newly created instance.

ContextData: Context-specific data required tocreate this service instance. Must conform to the schema specified by the ContextDataSchema.

InstanceKey: The URI of the new Instance resource that has been created. This is NOT the same as the key for the factory that is in the Response header.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
```

```
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:CreateInstance.Request>
      <aws:StartImmediately>Yes|No</aws:StartImmediately>
      <aws:ObserverKey>? URI </aws:ObserverKey>
      <aws:Name>? string </aws:Name>
      <aws:Subject>? string </aws:Subject>
      <aws:Description>? string </aws:Description>
      <aws:ContextData>
        <-- extensible element -->
      </aws:ContextData>
    </aws:CreateInstance.Request>
  </env:Body>
</env:Envelope>
```

Example 17 Factory resource CreateInstance method request

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response>
      <aws:Key>URI of the process definition receiving this request</aws:Key>
    </aws:Response>
  </env:Header>
  <env:Body>
    <aws:CreateInstance.Response>
      <aws:InstanceKey> URI </aws:InstanceKey>
    </aws:CreateInstance.Response>
  </env:Body>
</env:Envelope>
```

Example 18 Factory resource CreateInstance method request

```
<xsd:element name="CreateInstance.Request">
  <xsd:complexType>
    <xsd:sequence>
       <xsd:element name="StartImmediately" type="xsd:boolean"/>
        <xsd:element ref="ObserverKey" minOccurs="0"/>
        <xsd:element ref="Name" minOccurs="0"/>
        <xsd:element ref="Subject" minOccurs="0"/>
        <xsd:element ref="Description" minOccurs="0"/>
        <xsd:element ref="ContextData"/>
    </xsd:sequence>
</xsd:element>
<xsd:element name="CreateInstance.Response">
    <xsd:element name="InstanceKey"  type="xsd:anyURI"/>
</xsd:element>
```

Schema 10 Factory resource CreateInstance method

5.2.4 ListInstances

This method returns a collection of process instances, each instance described by a few important process instance properties.

`Filter`: Specifies what kinds of process instance resource you are interested in.

`FilterType`: indicates what language the filter is expressed in.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:ListInstances.Request>
      <aws:Filter filterType="nmtoken">?
         string
      </aws:Filter>
    </aws:ListInstances.Request>
  </env:Body>
</env:Envelope>
```

Example 19 Factory resource ListInstances method request

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response...>
  </env:Header>
  <env:Body>
    <aws:ListInstances.Response>
      <aws:Instance>*
        <aws:InstanceKey> URI </aws:Key>
        <aws:Name...>
        <aws:Subject...>
        <aws:Priority...>
      </aws:Instance>
    </aws:ListInstances.Response>
  </env:Body>
</env:Envelope>
```

Example 20 Factory resource ListInstances method response

```
<xsd:element name="ListInstances.Request">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Filter" type=”xsd:string”>
        <xsd:attribute name=”filterType” type=”xsd:NMTOKEN”/>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="ListInstances.Response">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="InstanceKey"/>
      <xsd:element ref="Name” minOccurs="0"/>
      <xsd:element ref="Subject” minOccurs="0"/>
      <xsd:element ref="Priority” minOccurs="0"/>
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Schema 11 Factory resource ListInstances method

5.3 Observer Resource

5.3.1 Observer Resource Properties

The Observer resource can receive events about the state changes of a service instance. An observer is expected to have a Key property and a PortType property.

`Key`: a URI that uniquely identifies this resource. All resources must have a Key property.

`PortType`: a list of the resource types that this resource supports.

```
<xsd:element ref="Key"/>
<xsd:element ref="PortType" maxOccurs="unbounded"/>
```

Schema 12 Observer resource properties

5.3.2 GetProperties

This method is the same as it was with Instance and Factory resources.

5.3.3 Completed

The Completed method indicates that the Instance has completed the work. This is the 'normal' completion.

This function signals to the observer resource that the started process is completed its task, and will no longer be processing. There is no guarantee that the resource will persist after this point in time.

`InstanceKey`: The URI of a process that is performing this work

`ResultData`: Context-specific data that represents the current values resulting from process execution. This information will be encoded as described in the section Process Context and Result Data above. If result data are not yet available, the ResultData element is returned empty.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:Completed.Request>
      <aws:InstanceKey> URI </aws:Instance>
      <aws:ResultData>
        <-- extensible element -->
      </aws:ResultData>
    </aws:Completed.Request>
  </env:Body>
</env:Envelope>
```

Example 21 Observer resource Completed method request

```
<?pseudo-xml?>
```

```
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response...>
  </env:Header>
  <env:Body>
    <aws:Completed.Response/>
  </env:Body>
</env:Envelope>
```

Example 22 Observer resource Completed method response

```
<xsd:element name="Completed.Request">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref=”InstanceKey”/>
      <xsd:element ref=”ResultData”/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="Completed.Response"/>
```

Schema 13 Observer resource Completed method

5.3.4 Terminated

This function allows the resource to communicate that is it abnormally ended for any of a number of reasons. The reason for the termination is passed as a parameter, but this is a descriptive value much like the exception values. The resource is not guaranteed to exist after this call. For that reason, the method request should include a dump of the properties of the terminating instance.

`Reason`: The reason that the process was terminated prematurely, if available

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:Terminated.Request>
      <aws:Reason> string </aws:Reason>?
      <-- Instance properties -->
    </aws:Terminated.Request>
  </env:Body>
</env:Envelope>
```

Example 23 Observer resource Terminated method request

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response...>
  </env:Header>
  <env:Body>
    <aws:Terminated.Response/>
  </env:Body>
</env:Envelope>
```

Example 24 Observer resource Terminated method response

```
<xsd:element name="Terminated.Request">
```

```
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:sequence>
        <xsd:element name=”Reason” type=”xsd:string” minOccurs="0"/>
      </xsd:sequence>
      <xsd:extension base=”instanceProperties”/>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
<xsd:element name="Terminated.Response"/>
```

Schema 14 Observer resource Terminated method

5.3.5 Notify

The Instance communicates to the observer via events. This is the method that is used to gives the other resource the event. This method used to send an event to a subscribed resource, particularly the Observer resource for a process instance.

The response to a notify event is not necessary. Typically, the header request tag will specify that no response is necessary.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:Notify.Request>
      <aws:Event.../>
    </aws:Notify.Request>
  </env:Body>
</env:Envelope>
```

Example 25

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response...>
  </env:Header>
  <env:Body>
    <aws:Notify.Response/>
  </env:Body>
</env:Envelope>
```

Example 26

```
<xsd:element name="Notify.Request">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Event"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="Notify.Response"/>
```

Schema 15

6 Data Encoding

6.1 Context data and result data

[more]

6.2 Size Limits and Data Resources

The HTTP protocol places no limit on the size of the result information for a particular request, but in practice if the data gets to be too large, there requests will tend to get very slow. AWSP has a built in assumption that all core data is returned with most requests to the process. As a practical limit, process instance should be designed to have a maximum of 64K of data total, and an average process instance handling only 5 to 10K bytes.

If the process instance needs to handle more data than this, then the larger pieces of data should be stored as separate documents on a HTTP server and passed as URI reference. In this case the tag which would normally carry the information is empty, and has an "xlink" attribute specifying the URI to the data.

If data is passed to a process instance as a reference, there is an implicit commitment that the data resource will remain accessible until the process instance is completed. If the process instance returns data as a reference, that data resource must remain accessible for until the process is completed, and then for the amount of time specified by the CleanupInterval property on the process definition.

6.3 Data Types

AWSP follows the SOAP and XMLSchema data type encoding specifications. The list of datatypes that MUST be supported by an AWSP client are:

- xsd:boolean
- xsd:integer
- xsd:string
- xsd:dateTime
- xsd:anyURI

6.4 Extensibility

Actual implementations of these resources may extend the set of properties returned. This document defines the required minimum set, as well as an optional set. Every implementation MUST return the required properties. The implementation may optionally return additional properties. Use of extended properties must be carefully considered because this may limit the ability to interoperate with other systems. In general no system should be coded so as to require an extended attribute. Instead it should be able to function is the extended properties are missing. Future versions of this specification will cover the adoption of new properties to be considered part of the specification.

6.5 PortTypes Property

Every resource is able to return a PortTypes element to indicate which operations it will respond to. We use the term “port type” rather than “resource type” in order to comply with the terminology used by Web Services Definition Language (WSDL) [10]. This is a mechanism to allow extending the set of operations in custom directions. New operations are assigned to a new group. The caller can discover that the resource will respond to a particular group or not in this manner.

In the current implementation, there are only three resource types: Factory, Instance, and Observer. These values appear as strings. A requester must ignore any group names it does not recognize.

```
<xsd:simpleType name="PortTypes">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Instance"/>
        <xsd:enumeration value="Factory"/>
        <xsd:enumeration value="Observer"/>
    </xsd:restriction>
</xsd:simpleType>
```

Schema 16 PortTypes

6.6 Status Property

The overall status of the asynchronous web service is defined by a status property value. This is a string value composed of words separated by periods. The status value must start with one of the seven defined values below, but the value can be extended by adding words on the end of the status separated by periods. The extension must be a refinement of one of the seven states defined here, such that it is not necessary to understand the extension. The intention is that these extensions may be proposed for future inclusion in the standard. The seven defined base states are:

`open.notrunning`: A resource is in this state when it has been instantiated, but is not currently participating in the enactment of a work process.

`open.notrunning.suspended`: A resource is in this state when it has initiated its participation in the enactment of a work process, but has been suspended. At this point, no resources contained within it may be started. (optional)

`open.running`: A resource is in this state when it is performing its part in the normal execution of a work process.

`closed.completed`: A resource is in this state when it has finished its task in the overall work process. All resources contained within it are assumed complete at this point.

`closed.abnormalCompleted`: A resource is in this state when it has completed abnormally. At this point, the results for the completed tasks are returned.

`closed.abnormalCompleted.terminated`: A resource is in this state when it has been terminated by the requesting resource before it completed its work process. At this point, all resources contained within it are assumed to be completed or terminated. (optional)

closed.abnormalCompleted.aborted: A resource is in this state when the execution of its process has been abnormally ended before it completed its work process. At this point, no assumptions are made about the state of the resources contained within it. (optional)

```
<xsd:simpleType name="stateType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value=”open.notrunning”/>
    <xsd:enumeration value=”open.notrunning.suspended”/>
    <xsd:enumeration value=”open.running”/>
    <xsd:enumeration value=”closed.completed”/>
    <xsd:enumeration value=”closed.abnormalCompleted”/>
    <xsd:enumeration value=”closed.abnormalCompleted.terminated”/>
    <xsd:enumeration value=”closed.abnormalCompleted.aborted”/>
  </xsd:restriction>
</xsd:simpleType>
```

Schema 17 stateType

6.7 Event Type

An Event is a state change that can occur in the AWS that is externally identifiable. Notifications can be sent to an observer in order to inform it of the particular event.

Time: the date/time of the event that occurred

EventType: One of an enumerated set of values to specify event types: InstanceCreated, PropertiesSet, StateChanged, Subscribed, Unsubscribed, Error. The event types correspond to the message types that the resource can receive.

SourceKey: The URI of the resource that triggered this event, usually an observer resource but perhaps the instance resource itself.

Details: A catchall element for containing any data appropriate.

OldState: The state of the instance resource before this event occurred.

NewState: The state of the instance resource before this event occurred.

```
<xsd:element name="Event">
  <xsd:element name="Time" type="xsd:dateTime"/>
  <xsd:element name="EventType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value=”InstanceCreated”/>
      <xsd:enumeration value=”PropertiesSet”/>
      <xsd:enumeration value=”StateChanged”/>
      <xsd:enumeration value=”Subscribed”/>
      <xsd:enumeration value=”Unsubscribed”/>
      <xsd:enumeration value=”Error”/>
    </xsd:restriction>
  </xsd:element>
  <xsd:element name=”SourceKey” type=”xsd:anyURI”/>
  <xsd:element name="Details" type=”xsd:anyType”/>
  <xsd:element name="OldState" type="StateType"/>
  <xsd:element name="NewState" type="StateType"/>
</xsd:element>
```

Schema 18 Event

6.8 History Type

In the case of a synchronous service, the execution of a service would normally be completely within a single transaction. Asynchronous services that have several activities will normally have several transactions. It is important to distinguish between a transaction that will normally be a single interaction with the Instance, and the execution of the Instance that may span any number of transactions. Each AWSP request to the Instance will have an identified user using standard HTTP authentication. Any interactions that causes a change in the Instance will probably need to be committed and saved after the interaction since there can be any amount of time before the next interaction. Some services will keep a record of these interactions, under whose authentication they ran, and what the results were so that all this can be reviewed later.

The GetHistory method returns the list of all events that have occurred on this resource. Every interaction with the service is a transaction and typical services will record every transaction in a history log. There is no requirement that the service keeps a history log, but if it does the history should be returned with this method.

`Filter`: The filter condition on the set of history items. If the filter is missing, then all history items that pertain to this Instance will be returned. The attribute filterType indicates what language the filter is expressed in.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Request...>
  </env:Header>
  <env:Body>
    <aws:GetHistory.Request>
      <aws:Filter filterType="nmtoken">?
    </aws:GetHistory>
  </env:Body>
</env:Envelope>
```

Listing 1 Instance resource GetHistory method request

If access to history information is not supported for the specific process flow resource, then the SOAP Fault element should contain the HistoryNotAvailable exception.

Because history is potentially lengthy, it is not included in the GetProperties method results. Instead it must be retrieved through the use of this method.

```
<xsd:element name="History">
    <xsd:element ref="Event" maxOccurs="unbounded"/>
</xsd:element>
```

6.9 Exceptions and Error Codes

All messages have the option of returning an exception. Exceptions are handled in the manner specified by SOAP 1.2. The header information should be the same, but in the body of the response, instead of having an AWSP element such as

GetProperties.Response or CreateInstance.Response, there will be the SOAP exception element env:Fault.

Multi server transactions: AWSP does not include any way for multiple servers to participate in the same transactions. It will be up to individual systems to determine what happen if a AWSP request fails; In some cases it should be ignored, in some cases it should cause that transaction to fail, and in some cases the operation should be queued to repeat until it succeeds.

```
<?pseudo-xml?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
  <env:Header>
    <aws:Response...>
  </env:Header>
  <env:Body>
    <env:Fault>
      <faultcode>env:Sender</faultcode>
      <faultstring>Header specific error</faultstring>
      <detail>
        <aws:ErrorCode>104</aws:ErrorCode>
        <aws:ErrorMessage>Invalid key</aws:ErrorMessage>
      </detail>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

Example 27 Exception

These error codes are chosen to be specific with the error codes defined by the Workflow Management Coalition Wf-MXL 1.1 specification. [Fit this is with SOAP Fault structure, improve the names since Fault uses string names.]

Header-specific	**100 Series**
These exceptions deal with missing or invalid parameters in the header.	
AWSP_PARSING_ERROR	101
AWSP_ELEMENT_MISSING	102
AWSP_INVALID_VERSION	103
AWSP_INVALID_RESPONSE_REQUIRED_VALUE	104
AWSP_INVALID_KEY	105
AWSP_INVALID_OPERATION_SPECIFICATION	106
AWSP_INVALID_REQUEST_ID	107
Data	**200 Series**
These exceptions deal with incorrect context or result data	
AWSP_INVALID_CONTEXT_DATA	201
AWSP_INVALID_RESULT_DATA	202
AWSP_INVALID_RESULT_DATA_SET	203
Authorization	**300 Series**
A user may not be authorized to carry out this operation on a particular resource, e.g., may not create a process instance for that process definition.	
AWSP_NO_AUTHORIZATION	301
Operation	**400 Series**

The operation can not be accomplished because of some temporary internal error in the workflow engine. This error may occur even when the input data is syntactically correct and authorization is permitted.

AWSP_OPERATION_FAILED	401

Resource Access	**500 Series**

A valid Key has been used, however this operation cannot currently be invoked on the specified resource.

AWSP_NO_ACCESS_TO_RESOURCE	501
AWSP_INVALID_FACTORY	502
AWSP_MISSING_INSTANCE_KEY	503
AWSP_INVALID_INSTANCE_KEY	504

Operation-specific	**600 Series**

These are the more operation specific exceptions. Typically, they are only used in a few operations, possibly a single one.

AWSP_INVALID_STATE_TRANSITION	601
AWSP_INVALID_OBSERVER_FOR_RESOURCE	602
AWSP_MISSING_NOTIFICATION_NAME	603
AWSP_INVALID_NOTIFICATION_NAME	604
AWSP_HISTORY_NOT_AVAILABLE	605

6.10 Language

AWSP messages should indicate their preferred language using the xml:lang attribute either in the SOAP Envelope element (the root element) or in the AWSP Request or Response element.

6.11 Security

HTTP provides for both authenticated as well as anonymous requests. Because of the nature of process flow in controlling access to resources, many operations will not be allowed unless accompanied by a valid and authenticated user ID. There are two primary means that this will be provided: HTTP authorization header or transport level encryption such as SSL.

The first and most common method of authentication over HTTP is through the use of the Authorization header. This header carries a user name and a password that can be used to validate against a user directory. If the request is attempted but the authentication of the user fails, or the Authorization header field is not present, then the standard HTTP error "401 Unauthorized" is the response. Within this, there are two authentication schemes:

- Basic involves carrying the name and password in the authorization field and is not considered secure.

- A digest authentication for HTTP is specified in IETF RFC-2069 [http://ietf.org/rfc/rfc2069.html], which offers a way to securely authenticate without sending the password in the clear.

Second, encryption at the transport level, such as SSL, can provide certificate based authentication of the user making the request. This is much more secure than the previous option, and should be used when high security is warranted.

Because the lower protocol levels are providing the user ID, AWSP does not specify how to send the client user ID. The authenticated user ID can be assumed to be present in the server at the time of handling the request.

Note that since most AWSP interactions are between programs that we would normally consider to be servers (i.e. process flow engine to process flow engine) the conclusion can be made that all such process flow engines will need a user id and associated values (e.g. password or certificate) necessary to authenticate themselves to other servers. Servers must be configured with the appropriate safeguards to assure that these associated values are protected from view. Under no circumstances should a set of process flow engines be configured to make anonymous AWSP requests that update information since the only way to be sure that the request is coming from a trustable source is through the authentication.

With the authentication requirements above, of either HTTP authorization header field or SSL secure transport, AWSP should be able to protect and safeguard sensitive data while allowing interoperability to and from any part of the Internet.

7 WorkList and WorkItem

There is no limit to the purposes that an Asynchronous Web Service may be put. This section, however, defines a particular web service that is of such great generality and utility that it is included here as both an example and as a suggestion for the most basic default service. While the intent of AWSP is to communicate between systems, there are times that steps in a process will be done more or less directly by humans. The worklist is a way to do this.

7.1 WorkList (Factory)

A WorkList is a special kind of factory that creates a WorkItem. Each user of the system will be associated with a particular WorkList so that all of their WorkItem resources will be created and managed by that factory. Users will be able to search and retrieve the WorkItem that are assigned to them from the WorkList using the ListInstances method.

7.2 WorkItem (Instance)

A WorkItem is a special kind of service instance that instead of performing something simply holds the information for an activity being performed by a person.

Just like all AWSs, in order to start an instance the factory must be presented with the context data. Since a factory must define an immutable schema, the schema consists of just a few general purpose properties that will be displayed to the user and/or returned in the results. AWSP does not include the ability to extend the schema of the instance to include new properties. Instead, a system interfacing to a worklist must compose text values that will make sense to a person reading them.

7.3 Workitem Context Data

Details: this property is the main body of the request. The systems interfacing to this worklist may have any number of separate properties holding details of the request. The values in all those properties relevant to the task at hand must be composed together in order to make a single readable presentation of the context data. Every AWS instance is

given a “Subject” and a “Description” which will describe in the normal way what is to be done. The “Details” field will hold all the rest of the information needed to do the task. For example, if the requesting process has an account number, or shipping addresses, they must be put into this ‘Details’ field for the user to see them.

Documents**:** this property holds any documents that are associated with the request.

Due Date**:** If a response is expected by a particular time, this date/time property will hold that value.

Answer**:** the process may be asking for an answer of some sort. The answer property give the user a place to enter this information that will, upon completion of the work item, be sent back to the requesting process. The value is text, and the format of this answer must be described in the Details field.

References**:** this property may contain a set of URIs that point to related materials that the user might find useful in completing the workitem.

External UI**:** this property holds an http URI that can be used to launch a browser to provide a UI for completing the activity.

7.4 Example

An example of how this might be used is

8 References

The following documents are relevant to this specification, and may be referenced in the text:

[1] Workflow Terminology (English), The Workflow Management Coalition, WFMC-TC-1011, version 2.0, June-1996, in PDF Format: [http://www.aiim.org/wfmc/standards/docs/glossary.pdf]. The terms used in this document are consistent with those found in this glossary.

[2] "Interoperability / Internet e-mail MIME Binding", The Workflow Management Coalition, WFMC-TC-1018, 1.1, July 1998 describes interoperation between workflow services using SMTP/MIME mail: [http://www.aiim.org/wfmc/standards/docs/if4-a.pdf.]

[3] "Reference Model", The Workflow Management Coalition, WFMC-TC-1003, 29-Nov-94, 1.1 describes the major components of a workflow system: http://www.aiim.org/wfmc/standards/docs/rmv1-16.pdf

[4] "HTTP - Hypertext Transfer Protocol" the latest information about HTTP can be found at [http://www.w3.org/Protocols/]

[5] "Hypertext Transfer Protocol -- HTTP/1.1", RFC-2068. the current proposed standard: http://www.w3.org/Protocols/rfc2068/rfc2068

[6] Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11 [http://globecom.net/ietf/std/std11.html], RFC 822 [http://globecom.net/ietf/rfc/rfc822.html] , UDEL, August 1982:

[7] "An Extension to HTTP: Digest Access Authentication" RFC-2069 January1997: http://www.w3.org/Protocols/rfc2069/rfc2069.txt

[8] “Simple Object Access Protocol (SOAP) Version 1.2 Part 1: Messaging Framework” W3C Working Draft 17 December 2001. http://www.w3.org/TR/soap12-part1/

[9] “Universal Description, Discovery and Integration (UDDI) Version 2.0 Programmer's API Specification” http://www.uddi.org/pubs/ProgrammersAPI-V2.00-Open-20010608.pdf

[10] “Web Services Description Language (WSDL) 1.1” W3C Note 15 March 2001. http://www.w3.org/TR/wsdl

[11] “Web Services Flow Language (WSFL)” May 2001. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[12] Universally Unique Identifier (UUID)

[13]” XML Schema Part 0: Primer”, W3C 2 May 2001, http://www.w3.org/TR/xmlschema-0/

[14] “XML Schema Part 1: Structures”, W3C 2 May 2001, http://www.w3.org/TR/xmlschema-1/

[15] Eric Steven Raymond, “The Cathedral and the Bazaar”, 24 Aug 2001. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/x188.html

9 Version History and Notes

2002.01.07 - Initial Draft

2002.02.03 - Another pass, include details, error code, XML Schema

2002.02.11 – Revision of terminology and reorganization of sections

10 Acknowledgements

A number of people have participated in the development of this document and the related ideas that come largely from earlier work:

Mike Marin, FileNET
Edwin Kodhabakchien, Collaxa Inc.
Dave Hollingsworth, ICL/Fujitsu
Marc-Thomas Schmidt, IBM
Greg Bolcer, Endeavors Technology, Inc
Dan Matheson, CoCreate
George Buzsaki and Surrendra Reddy, Oracle Corp.
Larry Masinter, Xerox PARC
Martin Adder
Mark Fisher, Thomson
David Jakopac and David Hurst, Lisle Technology Partners
Kevin Mitchell

Ian Prittie
Members of the Workflow Management Coalition

And many others...

11 Author Contact Information

Keith Swenson, MS2, kswenson@ms2.com

Jeffrey Ricker, Trans-enterprise Integration Corp., ricker@trans-enterprise.com

12 Open Issues

Remove this section before distribution