INTERNET-DRAFT
<draft-swenson-swap-prot-00.txt>
Expires February, 1999

K. Swenson
Netscape Communications Corp

August 7, 1998

Simple Workflow Access Protocol (SWAP)

Abstract

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

The Simple Workflow Access Protocol is a proposed way to solve this problem through use of HTTP protocol, and by transferring structured information encoded in XML. A new set of HTTP methods is defined, as well as the information to be supplied and the information returned in XML, that accomplish the control of generic asynchronous services. This has applicability to workflow because a workflow is itself an asynchronous service, and it also has the need to be able to call asynchronous services aoutside of itself.

This document will

1. give an Executive Overview
2. specify the goals of SWAP
3. explain how resource (object) model works and how URIs are used to

invoke methods of those resources.
4. explain how data is encoded to be sent or received.
5. specify preliminary details of the interface methods and parameters

1.0 Executive Overview

This protocol offers a way to start an instance of a generic asynchronous service, monitor it, control it, and be notified when it is complete. To clarify the discussion an instance of a service is called a process instance. This process instance can perform just about anything for any purpose. The key aspect is that the process 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 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 workflow? While this protocol is useful for starting and controlling process instances of any type, the need for this capability is of high demand in the workflow community. A workflow process is a process instance that may run for days, months, or even years, so this protocol will be useful for starting workflow processes. It is also the case that workflow services 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 emmensely useful to the workflow community, but has little actual specific to workflow.

How does it work?: You must start with the URI of a process definition. An HTTP request to this URI will cause this process definition to generate a process instance, and return the URI of this new service instance which is used for all the rest of the requests. The process instance can be provided with data (name/value pairs) by another HTTP request. The current state of the process instance can be retrieved with another HTTP request. The process instance can be paused or resumed with another HTTP request. There is also a pair of requests that may be used to give input data to the process instance, and to ask for the current value of the output data.

When it is done? The process instance runs asynchronously and takes whatever time it needs to complete. The originating program can, if it chooses, keep polling the state of the process 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 process instance with the URI of an observer. When the process instance is completed it will do an HTTP 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 process instance to complete.

1.1 Discussion of this Draft

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

swap-wg@netscape.com

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.ics.uci.edu/pub/ietf/swap/

2.0 Goals

2.1 Problem Statement

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

The protocol should be light weight 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 reimplemented a different way.
* The protocol should be consistent with the HTTP 1.1 extension mechanisms.
* 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 asynchrounous 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.
* Terseness of expression, or convenience for human readers, is not a goal of this protocol. Ease of generation and parsability should be favored over compactness.
* Like other internet protocols, SWAP should not require or make any assumptions about the platform or the technology used to implement the generic asynchronous service.

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 SWAP 1.0 does 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, SWAP 1.0 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. SWAP 1.0 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).
* SWAP 1.0 will not include the ability to perform maintenance of the process definition such as installation or configuration.
* SWAP 1.0 will not provide any capability to search or locate process instances that are running on the system. It will not support statistics or diagnostics of the running processes.
* SWAP 1.0 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 SWAP 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.

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 issue should not be part of SWAP working group meetings.

2.4 Audience

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

2.5 Terminology

The terminology used in this document borrows from the terms standardized by the Workflow Management Coalition. A copy of this glossary is freely available at: [1] http://www.aiim.org/wfmc/standards/docs/glossary.pdf. Important terms are listed below, some have refined definitions more suitable to this specification.

Process	The term "process" by itself is avoided in this

	document because it can be used several ways. Instead, the more precise terms "process definition" and "process instance" are used, which are defined below. The term process will only be used when the statement might apply equally to a process definition and a process instance.
Process Definition	This represents a "way of doing some work". A process 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 process definition is that while it embodies the knowledge of how a work it performed, it does not actually do the work.
Process Instance	This is the actual "performance of work". It embodies the context information that distinguishes one performance of a process from another. If a process definition represents a (software) program, then a process instance represents an invocation of that program. If the process definition represents a large workflow insurance claims process, then a process instance represents a single case of that process -- a particular insurance claim.

2.6 Related Documents

The Workflow Management Coalition (WfMC) has been working on the problem of workflow interoperability for many years. There are a number of WfMC documents which may be useful (but not required) in understanding the design rationale behind this document. Most related is the the [2] "Interoperability / Internet e-mail MIME Binding" document (WFMC-TC-1018) which describes how to allow interoperation between workflow services using SMTP/MIME mail. Another useful document to understanding how such services would fit together is the [3] Workflow Architural Reference Model where the basic components of a workflow system is described.

This specification can be loosly viewed as a binding of the Interface 4 abstract specification to the HTTP protocol. An understanding of the HTTP protocol and how it works is assumed in order to understand this document. Information on HTTP can be retrieve from: [4] http://www.w3.org/Protocols/. Particular, [5] RFC-2068 describes the HTTP/1.1 protocol used for SWAP can be found at http://www.w3.org/Protocols/rfc2068/rfc2068.

3.0 Resource Model

The generic service appears externally as a small number of different resources. The term resource is used instead of object in order to avoid discussion on whether the things being accessed are in fact OO style objects. The important aspect of a resource is that they are distinguished

by their web address, in the form of an URI. How these resources are implemented, whether using an OO language, or more traditional means, is not of concern to the protocol. What matters only is that when valid requests are made to valid URIs a proper and consistent result is returned.

There will be a small number of different types of these resources that will respond differently to different requests. In order to organize this document, the types of requests will be grouped together into interfaces. Saying that an resource implements an interface means that it responds appropriately to all of the requests contained by the interface. It will often be the case that a resource will implement 2 or more interfaces which means simply that all of those requests can be used to that URI address.

3.1 Process Instance / Process Definition / Observer

There are three fundamental interfaces that one needs to understand initially and which form the core of the resource model. These are the ProcessInstance, ProcessDefinition, and the Observer.

ProcessInstance: The process instance is the actual performer of the service. Every time the service is to be invoked, a new process instance is created. 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. 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 SWAP with a relatively simple CGI wrapper that implemented the process instance interface which would allow the transaction to be triggered by any tool that supported SWAP.

ProcessDefinition: Since each invocation of the service creates a new process instance resource, each with a unique URI, one needs a process definition which has a well known name in order to create the process instances. For each kind of service that is to be performed, a unique process definition URI is provided. Knowing the URI of the process definition is all that is needed in order to run the most basic services. Because the service generates a unique process instance resource for every invocation of the service, it allows many invocations to be performed at the same time, a very important feature when the service takes several months to complete.

Observer: The process instance resource knows very little about what invoked it, but it needs a way to communicate when it completes. The process instance can be given a reference to an observer resource. The only purpose of the observer interface is to provide a way for the process instance to report that it is finished. The URI of the observer resource is given to the process instance. When the perfomer is finished, it make a request to the provided URI to inform that it is completed. There can be

any number of observers on a particular process instance. Besides completion, there can be other occasions that will cause the perfomer to make a request to the observer to inform it of the change in state or data.

3.2 ActivityObserver

ActivityObserver: An ActivityObserver tells about external work that the process instance is waiting on. Simple process instances will be self contained, not relying upon any external process instances, and can be interfaced to the web without using the ActivityObserver interface. There is, however, the case where in order to complete its task, a process instance will call on external asynchronous services. The ActivityObserver resource is the way to indicate this. The ActivityObserver will describe what the process instance is waiting to have done, and will provide the URI to the process instance if there is one.

The ActivityObserver forms a bridge between the process instance that it is part of, and the sub-process instances that are doing work externally. It allows the currently open process instances to be browsed by a third party in both directions -- both from a parent process to a child process, and from a child process to a parent process.

For example, consider a process instance "Release Product", which is waiting on an activity observer "Approve Final User Manual", which is being performed by a process instance "Review Document". A person, lets call him Nick, might access "Release Product" process instance in order to determine why the product has not been released yet. The result will include the activity observer "Approve Final User Manual" which tells him what is needing to be done. That activity observer also has the URI of the performing process instance "Review Document" and uses this URI to find out the status of that child process instance. Similarly, a different person, let's call her Sonya, has been working on the approval of the user manual, and wants to find out why this document is being reviewed. By accessing the process instance Review Document" she can get the URI of the activity observer, and can see that it has a description "Approve Final User Manual" and can see that this is part of a process to "Release Product".

The question is often asked: "Why not cut out the activity observer resource, and allow browsing directly from parent process instance to child process instance." The answer is not apparent until you consider issues related to integrating two different workflow process definitions, designed by different people into a single combined process. The child process, in the example above, is a generic document review process which is being used for a user manual, but might be used for any number of different kinds of documents, and therefor has a very generic description of what is being done. The activity observer describes the exact same activity, except it does so from the point of view of the parent process, and as such helps to describe the relationship between the parent and child processes. Remember that the parent process may be waiting for any number of child processes at any given time. Retrieving from the parent process instance a list of child process instances directly may leave you with 5 instances of the

"Review Document" process.

The ActivityObserver resource also represents a view onto the process instance that is appropriate for the activity. The data retrieved from the activity reflects the schema of the invoked process.

3.3 WorkList / WorkItem

WorkItem: A WorkItem is a special kind of process instance that instead of performing something, simply represents that activity being performed by a person. The WorkItems hold the references to the activities for the people, who will search for and retrieve all of the WorkItems assigned to themselves

WorkItemFactory: A WorkItem, which is a Process Instance, is created by a WorkItemFactory, which is a Process Definition. Each user of the system will be associated with a particular WorkItemFactory so that all of their WorkItems will be created and managed by that factory. Users will be able to search and retrieve all of the WorkItems that are asigned to them from the WorkItemFactory.

3.4 URIs

Each resource has an URI address. This URI should be treated as an opaque value so that there is no requirements upon it. A given implementation of workflow 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 names, so that the names can server as a unique id for the resource involved. The receiving program should treat it as an opaque value an not assume anything about the format of the URI.

3.5 Method

Every interaction will specify a method of the request. The method is a standard HTTP header field that controls what the server does. The standard methods for retrieving web pages is GET. Methods used in this proposal include PROPFIND, PROPPATCH, GETHISTORY, etc.

3.6 Parameters

Associated with a method will be any number of named values which form the parameters of the method. These will be represented in XML in the body of the request. 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.

Part of standard http protocol includes the name and password of the person making the request. It is assumed that this is used to authenticate the user, and that the user name is available to the routines that are doing the processing, so that systems can be selective about whom they give data to, and can even tailor the response to the requesting user.

3.7 Returned Values

The result of request is a block of data encoded in XML. 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.

4.0 Protocol & Data Encoding

4.1 HTTP

HTTP/1.1 (RFC-2068) is a protocol that defines a simple way for two programs to exchange information. The protocol consists of a client program which 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 RFC-822 message format, which means that they consist of consist of a start-line, one or more header fields (also known as "headers"), an empty line (i.e., a line with nothing preceding the CRLF) indicating the end of the header fields, and an optional message-body.

Certain message may carry entities which can be understood as the "information payload" of the message. SWAP extends HTTP by specifying new methods and entities. An entity defines a set of entity headers and (optionally) an entity-body. The message-body is the entity-body after the transfer encoding has been applied.

4.2 Message Headers

General headers: HTTP defines a set of general header fields which are applicable to both request and response messages; these include Cache-Control, Connection, Date, Pragma, Transfer-Encoding, Upgrade, and Via. Since these headers are specific to the message, and not to the entity, SWAP will not make any special requirements on these headers.

Request headers: HTTP defines a set of header fields that are present in a request message. These include Accept, Accept-Charset, Accept-Encoding, Accept-Language, Authorization, From, Host, If-Modified-Since, If-Match, If-None-Match, If-Range, If-Unmodified-Since, Max-Forwards, Method, Proxy-Authorization, Range, Referer, User-Agent. SWAP will define new values for the method header. Standard methods OPTIONS, POST, PUT, DELETE, TRACE are not necessarily allowed by SWAP resources. ??? Methods GET and

HEAD are required to be supported but it is not clear at this time what Get should return. The rest of the headers, since they pertain to the message itself will not have any special requirements when used for SWAP.

Response headers: HTTP defines some standard response headers including: Age, Location, Proxy-Authenticate, Public, Retry-After, Server, Vary, Warning, WWW-Authenticate. Since these pertain to the response message, SWAP add no new requirements on these.

Entity headers: HTTP defines some standard entity headers: Allow, Content-Base, Content-Encoding, Content-Language, Content-Length, Content-Location, Content-MD5, Content-Range, Content-Type, ETag, Expires, Last-Modified. All SWAP messages exchange XML encoded content, which means that the following Content-Type value is required on SWAP messages:

* Content-Type: text/XML

4.3 XML Encoding

The actual entity data to and from the resource is a representation of a tree structure. Each node of the tree has any number of named attributes. The value of an attribute may be a simple value, or may be another node with more attributes. That tree of objects is encoded into tags with the tag name being the attribute name, and the value of the attribute between the start and end tags. Here is an example:

```
<swap>
    <interfaces>ProcessInstance</interfaces>
    <key>http://myServer/app1?proc=889</key>
    <validStates>
        <li>open.notRunning</li>
        <li>open.running</li>
        </validStates>
    <state>open.notRunning</state>
    <data>
        <d:city>San Francisco</d:city>
        <d:state>California</d:state>
        </data>
    </swap>
```

In the example above the root node has the following attributes: interfaces, key, validStates, state, and data. The attributes of interfaces, key, and state have simple values. The attribute validStates is a list of items, each item being a simple value. The data attribute has a list of fields each field expressed as a tag with the field name containing the field value (the "d:" is using the XML namespace mechanism to prevent these names from colliding with the names of the other tags -- all of the tags will in practice have such prefixes). This is an example of the sort of data that might be sent either to or from a resource. Each

method will specify what data needs to be sent to it, as well as what data it will send back. The specification of what data is needed for each method is described lower in the document. If a method returned the above data, then the specification would describe this using a bulleted outline such as this:

* interfaces: string
* key: the address of the resource
* validStates: a list of items
    - li: a single possible state value
* state: a string represeting the current state
* data: a list of fields that rpresent the data that has provided to the process instance
    - field: the name of the field

Custom tags will be invented for the known fixed attributes associated with standard objects and methods. XML gives you the ability to define new tags on the fly, and the parser can take those definitions in and extend the parser so that it can actually enforce the syntax as you define it. The order of the named tags do not matter. If an attribute is missing from the list, it is treated identically as if it were present with a null value, thus if a resource has a null value from an attribute it may simply be omitted from the encoded version. For strings, there is no distinction made between a null value and a zero length string; a null will be represented in the encoding as a zero length string.

4.4 Design Decisions about Methods

While the HTTP based protocol is light and easy to implement, the HTTP server is usually a stateless server which may need to load information about the resource which is being requested. Beacuse of this, a single large request is usually far more efficient than a series of small requests. SWAP has been designed to take this into consideration, and in general to to return as much information as possible in a single request. For this reason all resources have a PropFind method which returns all of the properties of an resource in a single call.

Furthermore, there are cases where an attribute on one resource returns a collection of other resources which could be accessed individually. In these cases, along with the URI for for the resource the resulting data set will include a certain number of other attributes for that resource. For example, a request for a list of ActivityObservers includes not only the URI (which is useful for doing PropFind with) but also the name, description, priority, etc. of the ActivityObserver. What this means is that in the initial listing you retrieve partial information about a resource. To get the complete information about a resource you have to perform the PropFind request on the URI. This spec must identify which attributes are included in the partial information.

4.5 Size Limits

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. SWAP 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 web server.

4.6 Exceptions

In all operations there will be an optional return value for the exception. This will generally be null or empty. The presence of a value will indicate that the operation did not complete successfully, and will contain a description of what when wrong. The description should not be assumed to be a simple string value, but is most likely to be a complex value created by combining several strings together. For translation purposes it is desirable to leave the message part of the exception separate from the data of the exception. The exception will have the following attributes:

* msg - this is the default (English) version of the error message. This is a required attribute
* exception - this exception (error) may have been caused by another, lower-level attribute. If so, the exception attribute contains this other lower level exception record. This can be seen as a linked list of exceptions, where the lower level errors are explained or refined by higher level error messages. This attribute is optional.
* other attributes - attributes can be added that are specific to the particular error message. For example, a "Can not open file" message might have an attribute of "filename" included in the exception record. These values are sent separate from the message so that a translated version of the message can be substituted for the English one, but the date values, like the "filename", can still be included in the report to the user.

Issue: Need to consider whether there should be a standard way to encode the substitution parameters into the error message. For example, a message like "Can not open file {filename}" the braces are used to indicate the place to substiture in the filename attribute. Java has a standard message formatting capability that involves braces and numeric indicators; for example: "Can not open file {0}" where the 0th parameter would be substituted in. In this case the attribute name would be "0" in the 'other attributes' above. This all sounds very reasonable, but should this be part of the standard specification in order for interoperability to succeed?

Issue: Should there be a unique ID attribute for the exception to make

lookup of localized versions of the message more efficient?

4.7 Extensibility

Actual implementations of these resources may extend the set of attributes returned. This document defines the required minimum set, as well as an optional set. Every implementation MUST return the required attributes. The implementation may optionally return additional attributes. Use of extended attributes 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 attributes are missing. Future versions of this spec will cover the adoption of new attributes to be considered part of the spec.

4.8 DataTypes

Since all data carried by XML must be encoded into a text format.

Date/time: All date/time values should be expressed in standard UTC format as specified by ISO 8601. A 24 hour clock is used, where the hour can range from 0 to 23. The year must be a 4 digit year, the month 2 digits, and the day two digits. An uppercase T separates the date from the time. All times are rounded to the nearest second. A date without a time value will use 00:00:00 as the specified time. An uppercase Z on the end indicates UTC time. This encoding is consistent with The IF4 MIME spec requirements for date/time encoding. Specifically, the following format must be used where the underlined parts have values substituted into them:

* yyyy-mm-ddThh:mm:ssZ

Numbers: Numeric values are expressed in text using standard ASNI-C literal expressions. The decimal separator for floating point numbers must be a period character (".").

Boolean: values are encoded with "1" representing true, and "0" representing false.

4.9 Security

HTTP provides for both authenticated as well as anonymous requests. Because of the nature of workflow in controlling access to resources, many operations will not be allowed unless accompanied by a valid and authenticated user id. There are two primary ways that this will be provided:

1. 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 which 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 RFC-2069 which offers a way to securly authenticate without sending the password in the clear.
2. 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 user id is being provided by the lower levels, SWAP 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 SWAP interactions are between programs that we would normally consider to be servers (i.e. workflow engine to workflow engine) the conclusion can be made that all such workflow 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 workflow engines be configured to make anonimous SWAP requests that update information since the 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, SWAP should be able to protect and safeguard sensitive data while allowing interoperability to and from any part of the internet.

5.0 Interface Documentation

Below each of the methods will be specified using a table such as this one. Each section is explained here:

Method:	This will be the name of the method and a short description of what it does.
Parameters:	* This is a list of the parameters for the method that are encoded into the body of the request. The order of these parameters is unimportant.
Result Page:	* This is a list of the return value incoded in the body of the result.

After the table will be more general explanation of the function and other things that should be known about it.

5.1 ProcessInstance Interface

The ProcessInstance interface must be implemented by all resources that

represent the execution of a long term asynchronous service, or in other words to perform work. The purpose of this interface is to allow the work to proceed asynchronously from the caller. The ProcessInstance represents a unit of work, and a new instance of the ProcessInstance 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, HTTP request 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 ProcessInstance, 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 ProcessInstance 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 which tell the requesting resource that the results are final and the ProcessInstance resource may be disappearing.

While a workflow process will implement ProcessInstance, it is important to note that there are also many non-workflow resources which will implement the ProcessInstance interface; it will also be implemented on any discrete task which needs to be performed asynchronously. Thus a wrapper for a legacy CICS transaction would implement the ProcessInstance interface so that that legacy application could be called and controlled by any program that speaks SWAP. A driver for an actual physical device, such as a numerical milling machine, would implement the ProcessInstance interface if that device is to be controlled by SWAP. Any program to be triggered by a workflow system that takes a long time to perform should implement the ProcessInstance interface, 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 ActivityObservers.

A ProcessInstance resource that is a workflow process is distinguished by the fact that it contains active ActivityObserver resources. A ProcessInstance can contain any number of ActivityObservers. These ActivityObservers represent requests for parts of the work to be performed, either from people or from other ProcessInstance resources.

Method:	PROPFIND - this is a single method that returns all the values of all the attributes of the resource.
Parameters:	* resultDataAttributes: (optional) this is a list of field names to specify which of all the result values are to be returned. If this is empty or missing than all result data values are included.
Result Page:	* interfaces: the names of the interfaces implemented on this resource. A resource may implement multiple interfaces. The purpose of this field is to tell

receives what kinds of requests can be made to the resource.

* name: The name attribute holds is a human readable identifier of the resource; the name of a resource is unique within the scope of its parent. The ProcessInstance name is automatically generated by the ProcessDefinition and is usually a simple unique integer.
* key: the proper URI of this resource. This is a globally unique. It is the one preferred URI for the resource.
* subject: a short description of this instance of the ProcessInstance. Sort of like the subject of an email message, it tells a little bit about this invocation of the service.
* description: a longer description of the instance of the ProcessInstance. To find a description of what the perfomer generically (as opposed to this instance) does get the description of the process definition.
* state: a string value that denotes the current state of the resource
* validStates: a collection of possible state values allowed by this resource.
* definition: the URI of the process definition resource.
* activities: a list of records containing for each active activity in the process:
    - URI: the URI of the active ActivityObserver
    - name: the name
    - state: the state
    - expirationDate: the (next) deadline date for this activity
    - assignees: a list of distinguished names of people assigned to this activity
    - creationDate: the date the activity became available
    - hasExpired: a boolean that states whether this activity is beyond a deadline.
* observer: the observer URI that should be used to notify about completion of the process instance, or other events. Optional.
* resultData: the set of name value pairs that represents the current result values; this list is not necessarily complete until the ProcessInstance is in a completed state. A list of items:
    - name:
    - value:
* priority: The priority of a ProcessInstance defines its relative importance. The values can range from 1 to 5 with 1 being highest priority (most important). Default value is 3. Optional.
* userInterface: the URI to the user interface of the

process instance. Launching a web browser on this URI would allow the user to view and manipulate the process instance. Optional.
* creator: the globaly unique distinguished name of the user that was authenticated in order to create the process instance. Optional.
* lastModified: The 1061 encoded date of the last modification. Optional.
* exception: if any, and if present, the above values may be empty

Method:	PROPPATCH - a uniform way to set any number of attributes of a resource simultaneously.
Parameters:	* subject: (optional) The short description of this instance of the resource. * description: (optional) a longer description of the instance of the resource * state: (optional) the new state to change to. The state is specified as a string which must be one of the allowed states of the resource. Not all states are reachable from the current state and so this operation may return an exception if the desired state is not reachable. * priority: (optional) an integer from 1 to 5, with 1 being highest priority. * data: (optional) a collection of name value pairs that represent the context of this ProcessInstance. The names of the fields 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 (e.g. a single name value pair) can be used to update just those fields in the partial set having no effect on fields not present in the call.; a list of items: o name: o value:
Result Page:	* returns the same thing that PropFind returns

All resources implement PROPPATCH 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 workflow 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. Subclasses will naturally allow for more attributes to be set. This returns the complete info for the resource, just as the PropFind method does, which will include any updated values. This behavior is defined to be the same as the PROPPATCH method defined in the WebDAV protocol.

Method:	TERMINATE - cancel the execution of the process. Used when the initiator is no longer interested in the service being performed.
Parameters:	* reason: a descriptive reason, if any, as to why this process is being terminated prematurely for recording in the log.
Result Page:	* exception: if any, and if present, the above values are empty

When a ProcessInstance is terminated that is waiting on other ProcessInstances, then it should terminate those ProcessInstances that is waiting on.

5.1.1 Notification.

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

Method:	SUBSCRIBE - This is a way for other implementations of the Observer interface to register themselves to receive posts about changes in process instance state. Not all ProcessInstance resources will support this; those that do not will return an exception value that will explains the error.
Parameters:	* observer: an URI to a resource that both implements the Observer interface and will receive the events.
Result Page:	* exception: if any

Method:	UNSUBSCRIBE - This is the opposite of the subscribe method. Resource removed from being listeners will no longer get events from this resource.
Parameters:	* observer: The URI of the resource to be removed from the listeners list. This must match exactly to an URI already in the list; if it does, then that URI will be removed; if not, then there will be no change to the process instance.
Result Page:	* exception: if any

5.1.2 Transactions

In the case of a synchronous service, the execution of a service would normally be completely within a single transaction. Asynchronous services which have several ActivityObservers will normally have several transactions. It is important to distinguish between a transaction which will normally be a single interaction with the ProcessInstance, and the execution of the ProcessInstance which may span any number of transactions. Each HTTP request to the ProcessInstance will have an identified user using standard HTTP authentication. Any interactions that

causes a change in the ProcessInstance 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.

Method: GETHISTORY - 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 keep a history log, but if it does the history should be returned with this method.

Parameters:
* filter: the filter condition on the set of history items. If the filter is missing, then all history items that pertain to this ProcessInstance will be returned.
* filterType: an value that indicates what language the filter is expressed in.

Result Page:
* history: a list of event objects, each event object contains:
    - timestamp: UTC time of the event
    - eventCode: an integer event code value. A list of valid event codes is defined for work processes, work activities and work items in their respective sections below. Event codes 0...255 are reserved for this purpose; vendor specific extensions may use event codes outside of that range.
    - eventType: a human readable description of the type of the event; for event codes in the reserved range the value of this attribute is
WfMF
.
    - responsible: the destinguished name of the participant that caused the event.
    - sourceKey: the URI of the resource that the event is referring to.
    - sourceName: the user friendly name of the source resource
    - containerKey: the URI of the containing resource, if any
    - oldState: if this was a state change event, then this is the old state
    - newState: if this was a state change event, then this is the old state
    - transition: if this was a state change event, the name of the transition taken.
    - changedData: if this was a dataChanged event, a collection of name/value pairs that represent the data items that were changed.
    - changedRole: if this was a ParticipantChangedEvent event, the name of the role that changed

        - participants: if this was a ParticipantChangedEvent event, the resulting list of distinguished names of the participants of the role.
    * exception: if any
        - the HistoryNotAvailable exception is raised if access to history information is not supported for the specific workflow resource.

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

Multi server transactions: SWAP 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 SWAP 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.

5.1.3 ProcessInstance States

ProcessInstance resources must implement the following states, as consistent with the WfMC process instance states:

* open.notRunning.notStarted - A workflow resource is in notStarted state when it has not yet actively participated in the enactment of a work process.
* open.notRunning.suspended - When a workflow resource is suspended, no workflow resources contained in this element may be started.
* open.running - In this state a workflow resource is performing its part in the overall execution of the workflow process.
* closed.completed - When a workflow resource has finished its task in the overall workflow process it enters the completed state; it is assumed that all workflow resources contained in that element are completed when it enters this state.
* closed.terminated - Execution of a workflow resource may be terminated before it completes its task. It is assumed that all workflow resources contained in this element are either completed or are terminated when it enters this state.
* closed.aborted - A workflow resource may be aborted before it completes its task (or even before it ever started performing its task). No assumptions on the state of workflow resources contained in this element are made when it enters this state.

5.2 ProcessDefinition Interface

The interface for resources that can create ProcessInstances. The description of a process definition is the description of what the ProcessInstance that it creates does.

Method: PROPFIND - this is a single method that returns all the values of all the attributes of the resource.

Parameters:
* (none)

Result Page:
* interfaces: the names of the interfaces implemented on this resource.
* name: The name attribute holds is a human readable identifier of the resource; the name of a resource is unique within the scope of its parent. In some cases the name may be nothing more than a unique number.
* key: the proper URI of this resource. This is a globally unique. It is the one preferred URI for the resource.
* description: a longer description of this process definition
* state: a string value that denotes the current state value.
* validStates: a collection of possible state values allowed by this resource
* contextDataInfo: meta-data information about what name values pairs are expected to be put into this process at start time; a list of items:
    - name:
    - type:
* resultDataInfo: meta-data information about what name values pairs are expected to be returned by this process upon completion; a list of items:
    - name:
    - type:
* exception: if any, and if present, the above values may be empty

Method: CREATEPROCESSINSTANCE

Parameters:
* observer: the URI to the Observer resource. If present, then the process instance is expected to report state changes, especially the transition to completed state, to this resource
* name: the name of the process instance to create. This must be unique for all of this kind of ProcessInstance. Process Definitions will attempt to use this value but if not unique, they will either modify the value until it is unique, or else generate a new value, so the use of this value can not be counted upon
* subject: a short description of why the ProcessInstance is being created
* description: a longer description of the ProcessInstance.
* contextData: a collection of name/value pairs:
    - name:
    - value:

* startImmediately: (optional) a boolean "yes" or "no" that specifies whether the newly created process should be started immediately. If not started, then the process is in an "initial" state, and you have to call the start operation on it. Default is "yes"

Result Page:
* key: The URI of the new ProcessInstance resource that has been created.
* exception: if any, and if present, the above values are empty

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.

Method: LISTINSTANCES - return a collection of process instances, each instance described by a few important process instance attributes.

Parameters:
* filter: a filter to specify what kinds of process instance resource you are interested in.
* filterType: a value that indicates what language the filter is expressed in.

Result Page:
* instances: a list of values each containing
  - key: the URI of the ProcessInstance resource
  - name: the display name
  - priority: the priority
* exception: if any, and if present, the above values are empty

5.3 Observer Interface

This is an abstract interface for resources that can request work to be done, and can receive events of their state changes.

Method: PROPFIND - gets all the attribute values for the resource.

Parameters:

Result Page:
* interfaces: the names of the interfaces implemented on this resource.
* key: the proper URI of this resource. This is a globally unique. It is the one preferred URI for the resource.
* data: (optional) A resource implementing the observer interface may return the context data, which is a collection of name value pairs which are process attributes. The distinction between the context and the instanceData is that the context may have shifts in the names of the attributes to accomodate the sub process. In otherwords, the context is in the schema defined by the subprocess.

    - name
    - value
  * performer: the list of URIs of the ProcessInstance(s) of this Observer.
  * exception: if any, and if present, the above values may be empty

Remember that the implementation of the resource may cause it to return other values as well; PropFind always returns the union of all the properties specified by all the interfaces that the resource implements. The property above is a minimum set.

Method:	PROPPATCH - a uniform way to set any number of attributes with a single operation. All of the parameters are optional, but of course at least one must be present for this method to do anything.
Parameters:	* ProcessInstance: the URI of a process that is performing this work, if it exists. This is needed if there is more than one ProcessInstance. * resultData: (optional) a list of name value pairs that are the result data from the ProcessInstance.
Result Page:	* exception: if any

This is how the subprocess can communicate changes in the data back to the observer resource before the time of completion, in order to keep the attribute values of the two processes synchronized.

Method:	COMPLETE - indicate that the ProcessInstance has completed the work. This is the 'normal' completion.
Parameters:	* ProcessInstance: the URI of a process that is performing this work, if it exists. This is needed if there is more than one ProcessInstance. * exit: (optional) This tells which exit point was reached * resultData: a collection of name/value pairs that represent the final set of data as of the time of completion
Result Page:	* exception: if any, and if present, the above values are empty

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.

Method:	TERMINATED - This is how the ProcessInstance indicates that it is no longer running because someone has terminated it before it reached completion.
Parameters:	* reason: (optional) the reason that the process was terminated prematurely, if available
Result Page:	* exception: if any, and if present, the above values are empty

This function allows the subprocess 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.

Method:	NOTIFY - this method used to send an event to a subscribed resource, particularly the Observer resource for a process instance.
Parameters:	* eventObject: The event object contains a number of parameters, which may be extended by systems. o timestamp: the UTC time that the event fired o eventCode: o eventType: one of an enumerated set of values to specify event types. o sourceKey: o sourceName: o containerKey: o oldState: (optional) for a state change event, this is the original state. o newState: (optional) for a state change event, this is the new state. o transition: (optional) for a state change event, this is the transition that was used. o changedData: (optional) for a data changed event, this specified the data that has changed, which is a list of items: + name + value o role: (optional) for a role changed event this specifies the role. o participants: (optional) a list of participants that are now assigned to the role.
Result Page:	* exception: if any, and if present, the above values are empty

The ProcessInstance communicates to the observer via events. This is the method that is used to gives the other resource the event.

5.4 ActivityObserver Interface

The ActivityObserver is not a performer of work, but rather the requester, and subsequent observer of work. Instantiation of a ActivityObserver resource is the way to ask a workflow system to perform work. It may internally have many levels of representation of that work, breaking it into numerous tasks and subtasks. Those internal levels are not the subject of this version of standardization (but are good candidates for version 2 of this spec.) Eventually the workflow system breaks the tasks out to the point where it needs to request work to be performed by a exernal resource. By external, we mean that the request will be

coordinated purely by the standard mechanisms outlined in this specification. In order for a workflow system to coordinate externally performed work, it must have an instance of a resource that implements the ActivityObserver interface.

ActivityObserver extends the Observer interface; it is a request for work to be done which is part of a bigger process. A ActivityObserver does not have to be performed by a subprocess, but can be performed by people using a user interface that calls methods on the ActivityObserver resource directly, for instance to indicate that the activity is completed, or to send the result data values. A ActivityObserver resource provides a way to browse up to the ProcessInstance that it is contained by.

The primary purpose of the ActivityObserver resource is to indicate what activities are currently being done within a process. Given a reference to the ProcessInstance resource, a method will return a collection of the currently active activities. From the ActivityObserver, one can discover the sub ProcessInstance resources if any exist, which may contain more current activities. In this way, distributed workflow of any scale can be navigated.

When a ActivityObserver is completed (it is told that the work is complete) then the workflow system, through the use of internal logic, determines what new activity is enabled. It is important to note that this specification does not include a definition of how that logic is configured or how it executes; this will presumably be the subject of future specifications. What matters though is that a ProcessInstance communicates to an activity resource.

It is the responsibility of the ActivityObserver to conform to the interface required by the ProcessInstance that is performing the work. This means that the context data will be supplied to the ProcessInstance using the fieldnames and meanings that it expects. It means that as the ActivityObserver resource receives data changed events, that data will be expressed with the set of fields defined by the performing ProcessInstance. The overall effect of this is that the ProcessInstance can be implemented without any knowledge of the resources that will call it; but the ActivityObserver must have a certain minimal knowledge of the process it is calling. The ProcessDefinition interface supplies that knowledge when requested.

Method:	PROPFIND
Parameters:	* (none)
Result Page:	* interfaces: the names of the interfaces implemented on this resource. * name: The name attribute holds is a human readable identifier of the resource; the name of a resource is unique within the scope of its parent. In some cases the name may be nothing more than a unique number. * key: the proper URI of this resource. This is a globally unique. It is the one preferred URI for the

resource.
* description: a longer description of the instance of the resource
* state: a string value that denotes the current state value.
* states: a collection of possible state values allowed by this resource
* ProcessInstance: If the activity has a ProcessInstance (sub-process) associated with it, then the URI of the ProcessInstance is listed here
* container: this is the URI of the ProcessInstance resource which contains this work activity.
* priority: like the priority on a ProcessInstance.
* contextData: The data as appropriate for this activity, a list of item:
    - name:
    - value:
* assignees: a list of distinguished names of the person or people assigned to this ActivityObserver.
* conclusions: a list of conclusion values which may be used in the completed operation; these are the ways that an activity can be completed.
* userInterface: the URI to a page which is the user interface for this resource, if one exists. This URI is accessed using the GET method.
* expirationDate: the (next) deadline date for this activity. Presumably, when this date/time is reached the system will do something, like escalate the activity, or continue the process without this activity. Optional.
* hasExpired: a boolean value that tells whether this activity is currently beyond a deadline date. Optional.
* creationDate: the date this activity became available.
* conclusions: a list of values that can be used in the completed operation in order to indicate how the activity was completed. For instance may systems ask the user to specify "approved" or "not-approved". These conclusions would be listed in this attribute. It is important to note that completing an activity with "not-approved" is not the same as terminating the activity. Thr former indicates that the process is fine, but the subject needs rework, where the latter (terminate) might indicate that the process is broken.
* exception: if any

.

Method: PROPPATCH

Parameters:
* priority: the priority of the work process.
* resultData: the data that resulted from the performing of the activity. This can be a partial set of the

data; a list of items:
o name:
o value:

Result Page:
* (same results as PropFind)
* exception: if any

Method: COMPLETE

Parameters:
* data: (optional) the set of name value pairs that represent the result value of this work.; a list of items:
o name:
o value:
* option: (optional) The completion option that was chosen to complete the activity

Result Page:
* exception: if any, and if present, the above values are empty

This is how the workflow system is told that a work item has been completed

5.5 Servlet Interface

The servlet interface is not a necessary part of the SWAP protocol, but is included here as a convenience for having a consistent way to package up parameters and call a synchronous service via HTTP. This interface is based on that of the Java Servlet interface. A generic synchronous resource can implement the servlet interface, and a remote system will be able to pass it parameters encoded in XML, invoke the operation, and get the results encodedin XML. There is only one method: RUN. Each function that you call will have its own Servlet resource with its own URI.

Method: RUN

Parameters:
* (depends upon the function being called)

Result Page:
* (depends upon the function being called)
* exception: if any

* Issue: a search should be made to determine if this has already been defined in a different spec

5.6 WorkList Interface

The worklist is a ProcessDefinition, so it inherits all those functions. Creating a workitem done by using the standard createProcessInstance function. A list of workitems is retrieved using the listAllProcessInstances method. The only thing unique about a WorkList is that the ProcessInstances that it creates do not actually do anything, but instead represent a person doing it. They make it possible for the person to search for an find all the workitem assigned to that person.

The scheme is that for each user, the user directory would list the URI of the worklist server for that user. A workflow system that assigns an activity to a user, would look for the worklist server for that user, and then create a workitem (ProcessInstance) that matches up with the activity.

5.7 WorkItem Interface

The ProcessInstances that are created (called workitems) are a little different than normal ProcessInstances. Normally a ProcessInstance is programmed to perform a particular task, and it can give you information about that task. The workitem does not have any preprogrammed activity (other than simply "doit"). It instead picks up the description of the task from the activity, which is something that other ProcessInstances rarely if ever do.

WorkItems point a user to an activity. They allow the user a place to store the state of the activity, for example if the activity is paused this could be indicated int he status of the workitem. The user does not interact with the workitem other than to find the activity. Given the URI of the activity, the user deals directly with the activity. When the activity is completed, the workflow system terminates the workitem, causing it to disappear from the ther user's worklist.

5.8 EntryPoint Interface

The EntryPoint is a resource that describes a particular way that a process instance may be started. The ProcessDefinition resource provides a collection of EntryPoints. Each EntryPoint may have different requirements of data to be provided. Access to an entry point can be controlled by the engine, for example an organization can have an 'internal' EntryPoint that allows some fields to be specified, and an 'external' EntryPoint which is more restrictive. While EntryPoints provide some powerful capabilities for starting process instances, they are not required, since process instances can be started directly on the ProcessDefinition object.

Method:	PROPFIND - this is a single method that returns all the values of all the attributes of the resource.
Parameters:	* (none)
Result Page:	* interfaces: the names of the interfaces implemented on this resource. * name: The name attribute holds is a human readable identifier of the resource. * key: the proper URI of this resource. This is a globally unique. It is the one preferred URI for the resource. * description: a longer description of this process definition * conclusions: A list of 'conclusion' values that can be used in the create process instance command. * definition: the URI of the process definition.

* contextDataInfo: meta-data information about what name values pairs are expected to be put into this process at start time; a list of items:
    - name:
    - type:
* resultDataInfo: meta-data information about what name values pairs are expected to be returned by this process upon completion; a list of items:
    - name:
    - type:
* exception: if any, and if present, the above values may be empty

The EntryPoint does not have any properties that can be set on it. Furthermore, there is only a single operation, CreateProcessInstance.

Method:	CREATEPROCESSINSTANCE

Parameters:	* observer: the URI to the Observer resource. If present, then the process instance is expected to report state changes, especially the transition to completed state, to this resource
	* name: the name of the process instance to create. This must be unique for all of this kind of ProcessInstance. Process Definitions will attempt to use this value but if not unique, they will either modify the value until it is unique, or else generate a new value, so the use of this value can not be counted upon
	* subject: a short description of why the ProcessInstance is being created
	* description: a longer description of the ProcessInstance.
	* contextData: a collection of name/value pairs:
	  - name:
	  - value:
	* startImmediately: (optional) a boolean that specifies whether the newly created process should be started immediately. "0" means don't start, while a "1" means start immediately. If not started, then the process is in an "initial" state, and you have to call the start operation on it. Default is "1"

Result Page:	* key: The URI of the new ProcessInstance resource that has been created. * exception: if any, and if present, the above values are empty

---

6.0 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. Also note that glossaries in French and (soon) German are available at the same web site.

[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, RFC 822, UDEL, August 1982: http://www.w3.org/Protocols/rfc822/rfc822.txt

[7] "An Extension to HTTP : Digest Access Authentication" RFC-2069, January 1997: http://www.w3.org/Protocols/rfc2069/rfc2069.txt

7.0 Version History and Notes

7. 1 Meeting May 4-5

This document was circulated for preliminary review with the goal of meeting and discussing the contents on May 4-5 in Costa Mesa California. The following URI was made available for up to date information about this effort: http://www.ics.uci.edu/pub/ietf/swap/

7. 2 Update for WfMC meeting, May 6

The following changes were made to reflect the results of the May 5 meeting:

* changed the term "performer" to be "process instance"
* changed the term "performerFactory" to be "process definition"
* changed the term "WaitActivity" to "ActivityObserver"
* Expanded the explanation of the ActivityObserver and why it exists.
* changed "command" to method and rewrote the description
* updated the XML encoding section to reflect new information about this

* changed "getInfo" to PropFind to be the method to get properties
* receiveEvent changed to NOTIFY to conform with proposed notification service.
* addListener and removeListener changed to SUBSCRIBE and UNSUBSCRIBE
* removed the ProcessDirectory object as it is not needed.
* added Servlet interface for synchronous interactions

7. 3 Update for July Version

* changed URL to URI
* changed PROPSWITCH to PROPPATCH
* changed object to resource
* added references, terminology, related documents
* new section on HTTP, messages, and header fields
* added a section on security

8.0 Acknowledgements

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

Marc-Thomas Schmidt, IBM
Dan Matheson, CoCreate
Mike Marin, FileNET
George Buzsaki, Oracle Corp.
Surrendra Reddy, Oracle
Larry Masinter, Xerox PARC
Martin Adder
Mark Fisher, Thomson
Members of the Workflow Management Coalition
And many others...

9.0 Copyright

9.0 Author Contact Information

Keith Swenson
Netscape Communications Corporation
501 E. Middlefield Rd.
San Jose, CA, 94043
kswenson@ms2.com