Object Eventing Service

Draft Revision 1

Michael Boyle (boylem@cpsc.ucalgary.ca), and
Michael Rounding (rounding@cpsc.ucalgary.ca)

University of Calgary

Abstract

The Object Eventing Service provides a hierarchy of named nodes containing arbitrary binary data intended to support the development of distributed or collaborative applications. The service fulfills two key needs of users (e.g., developers of distributed or collaborative software):

- persistent data storage to be shared among a distributed collection of peers; and,
- selective notification of changes to the tree or the values stored at nodes.

Although this draft revision describes only a small fraction of the envisaged system, an effort has been made to recognize opportunities for further enrichment of the service and design has proceeded with the expectation that some or all of these enrichments may be added at a later date.

Table of Contents

Basic Concepts

Tree

The tree in OES is a hierarchically organized global namespace; *nodes* in the tree store arbitrary binary data. Every node in the tree has exactly one ancestor, with the exception of a single root node in the tree, which has no ancestors. A node may have *n*≥0 children; each child node has a *nodename* that is locally unique among its *n*-1 siblings. The nodename of any non-root node is a character string at least one character in length. The nodename of the root node is a zero-length character string. Figure 1 shows a hypothetical tree representing a few continents, nations, and cities of the world.

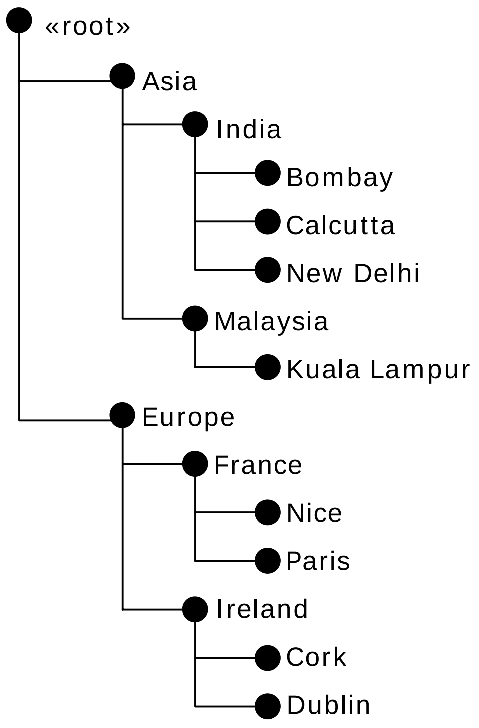

Figure 1. Hypothetical OES tree.

For this revision, UCS-2—also known as Unicode™—is assumed to be the character set employed in OES. For each node in the tree, there exists a single path of nodes extending from the root of the tree to that node. The concatenation of the names of each node along a path descending from the root of the tree, interleaved with a path separator character, constitutes the *pathname* of the terminus of the path. For this revision, the path separator character is assumed to be the reverse solidus (\) character (UCS-2 character code U+005E; commonly referred to as the 'backslash character'). Consequently, the legal set of characters that may appear in a nodename is UCS-2 less the path separator character. For example, the pathname of the node named 'Calcutta' is '\Asia\India\Calcutta'.

In this way, the tree resembles the familiar metaphor employed in file systems, with folders and files in the file system relating to nodes in the OES tree. Unlike the file system metaphor, however, non-leaf nodes in the OES tree may also store data: in OES, there is no distinction between a container node (i.e., folder) and a non-container node (i.e., file).

Notifications

The set of possible operations that may be made to a tree includes, but is not limited to:

- creating a node;
- removing a node; or,
- changing the value stored at a node.

We presume that distributed or collaborative applications which use OES will be interested in taking some appropriate action when one of these operations is performed. To fill this need, OES generates *notification events* for every such operation.

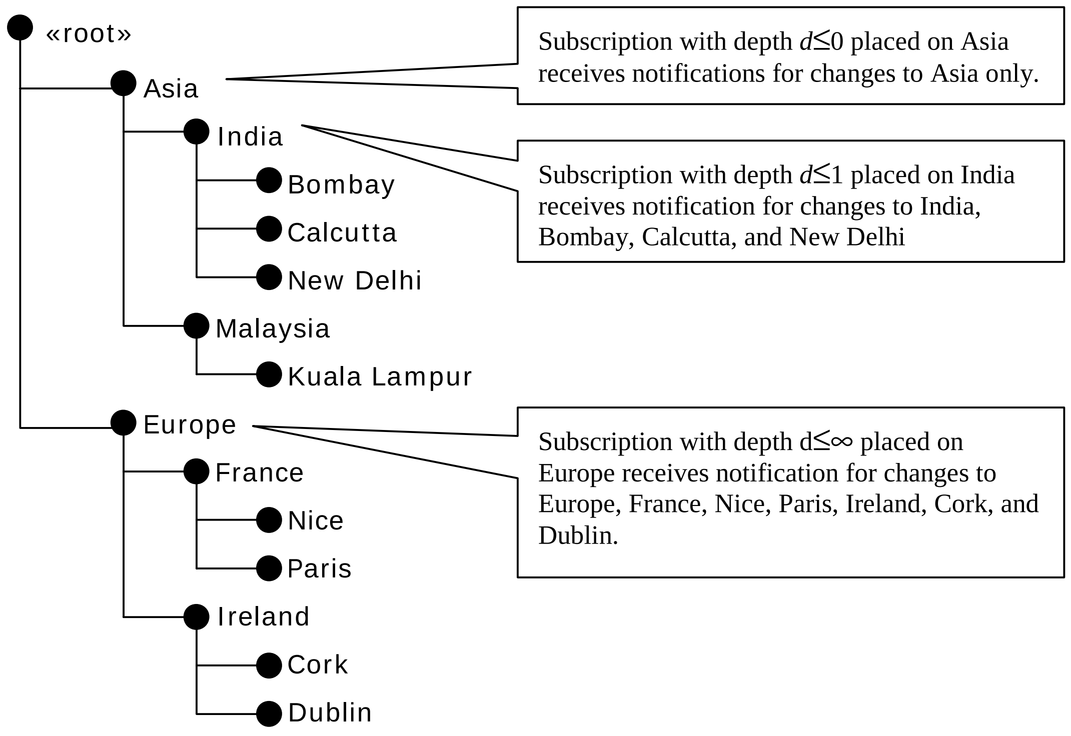
«root»
Asia
India
Bombay
Calcutta
New Delhi
Malaysia
Kuala Lampur
Europe
France
Nice
Paris
Ireland
Cork
Dublin
Subscription with depth $d \leq 0$ placed on Asia receives notifications for changes to Asia only.
Subscription with depth $d \leq 1$ placed on India receives notification for changes to India, Bombay, Calcutta, and New Delhi
Subscription with depth $d \leq \infty$ placed on Europe receives notification for changes to Europe, France, Nice, Paris, Ireland, Cork, and Dublin.

Figure 2. Subscrption depth.

Applications interested in receiving all of the notification events generated for any node in a particular sub-tree place a *subscription* at the root node of that sub-tree. The subscription includes information about the *depth* of the sub-tree rooted at the node to which to watch for changes. The depth is a non-negative integer. For example, in Figure 2, if a process places a subscription with depth zero on a the node named ‘Asia’, then it will only receive notification events for operations that affect ‘Asia’ directly. If the process places a subscription with depth one on the node named ‘India’, then it will receive notification events for operations that affect that node, plus its immediate children. If the process places a subscription with infinite depth on the node named ‘Europe’, then it will receive notification events for operations that affect any node in the sub-tree rooted at the node upon which the subscription was placed. Consequently, subscriptions are evaluated not just on the node where an operation takes place, but also all of the ancestors that lie on its path.

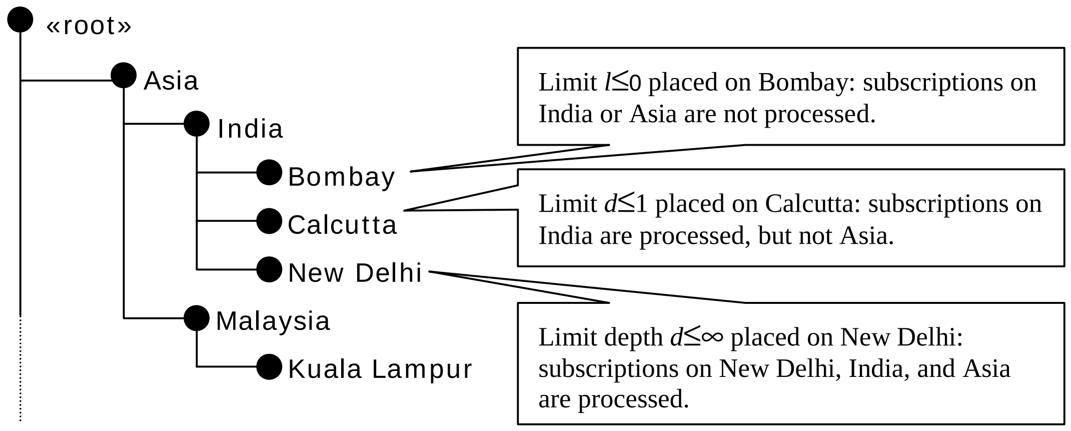
«root»
Asia
India
Bombay
Calcutta
New Delhi
Malaysia
Kuala Lampur
Limit $l \leq 0$ placed on Bombay: subscriptions on India or Asia are not processed.
Limit $d \leq 1$ placed on Calcutta: subscriptions on India are processed, but not Asia.
Limit depth $d \leq \infty$ placed on New Delhi: subscriptions on New Delhi, India, and Asia are processed.

Figure 3. Subscription notification upward propagation limit.

We presume that these operations may take place very frequently at some leaf nodes in the tree. To balance the desire to provide a simple programmatic interface and yet provide reasonable performance and ensure scalability, we introduce an *upward notification propagation limit* for each node. This limit is a non-negative integer and controls how far up the tree subscriptions are evaluated. For example, in Figure 3, if the limit on the node named 'Bombay' is zero, then only subscriptions placed directly on that node are evaluated. If the limit on the node 'Calcutta' is one then subscriptions placed on that node and subscriptions placed on that node's immediate ancestor ('India') are evaluated. If the limit on the node 'New Delhi' is infinity, then subscriptions placed on any node along the path leading to will be evaluated: 'New Delhi', 'India', 'Asia', and the root.

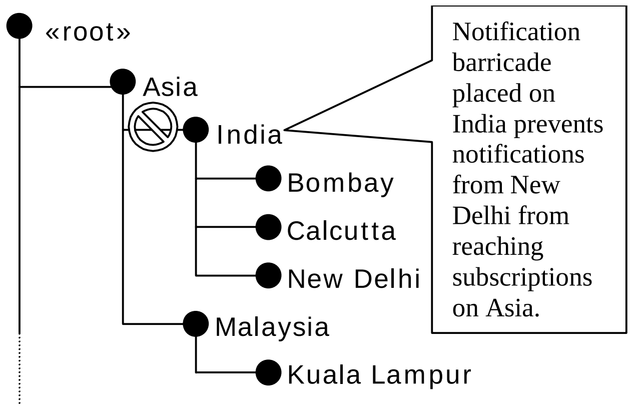
«root»
Asia
India
Bombay
Calcutta
New Delhi
Malaysia
Kuala Lampur
Notification barricade placed on India prevents notifications from New Delhi from reaching subscriptions on Asia.

Figure 4. Notification barricade.

To further improve scalability and simplify programming, two additional concepts are being considered: *notification barricades* and *subscription pools*. A notification barricade placed at a node prevents any notification from propagating upward beyond that node, regardless of the notification propagation limit of the node that generated the notification event. For example, in Figure 4, if a barricade is placed at the node named 'India' then notifications from the node named 'New Delhi' will no longer make it to subscribers on the node 'Asia.' Notifications from the nodes 'Bombay' and 'Calcutta' are unaffected by the barricade, as their respective upward propagation limits prevented notifications from these nodes from reaching 'Asia' anyway.

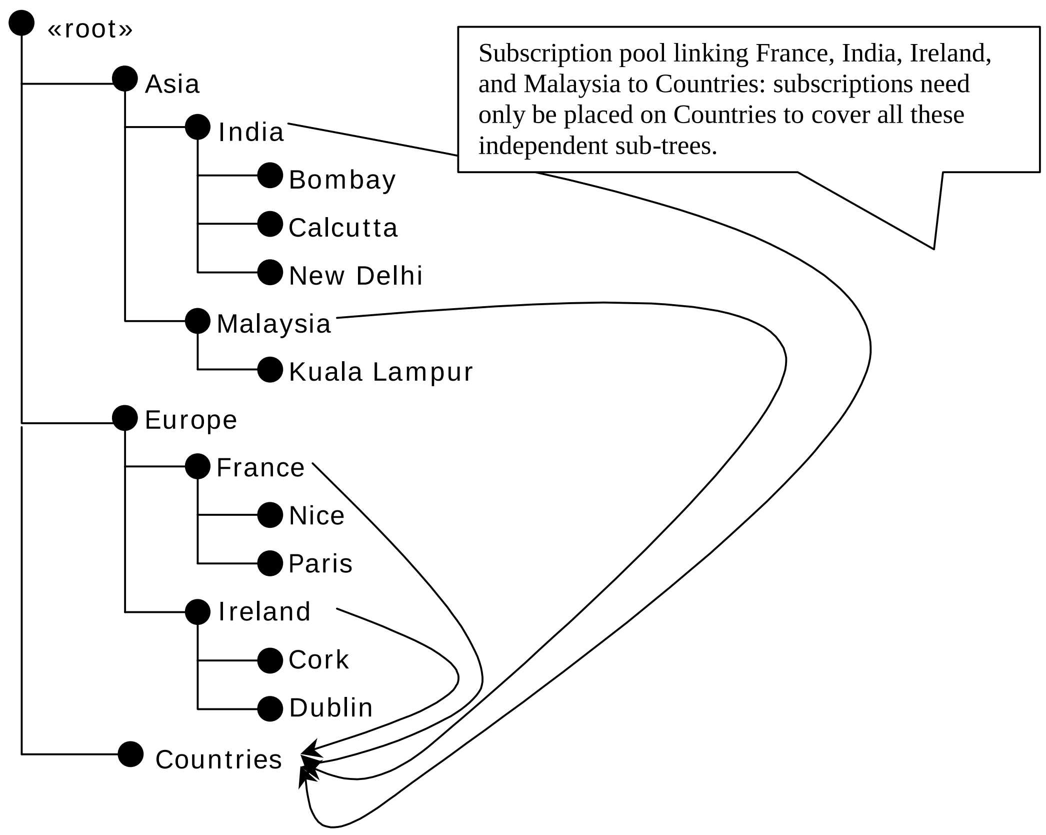
«root»
Asia
India
Bombay
Calcutta
New Delhi
Malaysia
Kuala Lampur
Europe
France
Nice
Paris
Ireland
Cork
Dublin
Countries
Subscription pool linking France, India, Ireland, and Malaysia to Countries: subscriptions need only be placed on Countries to cover all these independent sub-trees.

Figure 5. Subscription pools.

A subscription pool placed on a node short-circuits the normal upward bubbling of a notification event generated at that node, and redirects so that it appears it was generated from a different node. Normal bubbling of the redirected notification event continues upward from this other node. For example, in Figure 5, a subscription pool placed on France, India, Ireland, and Malaysia, redirecting to the node named 'Countries' means that changes to any of France, India, Ireland, or Malaysia result in subscriptions on the node name Countries to be processed. Subscription pools provide a convenient mechanism by which to design notification topologies that do not conform to the strict hierarchical model imposed by OES data store.

Subscription Contracts

A process *P* running on machine *M* may request that OES deliver to it notification events for an arbitrary collection of nodes. To facilitate this, the process *leases* a *subscription contract* from OES. The contract describes at least the means by which notification events are to be delivered and the time at which the lease expires. It is permissible for a process to renew its lease on a subscription contract, further extending into the future the time at which the lease expires. If the lease expires, however, all subscriptions associated with the expired contract are immediately cancelled, and the process, which took out the lease, will no longer receive any further notifications associated with that contract. (It may continue to receive notifications, in general, if it had taken out other contracts that have not yet expired.) Moreover, all memory of the contract (e.g., the set of subscriptions made with it) is cleared and its subscription-contract-id is rendered meaningless.

Quality of Service

We presume that a variety of qualities of service will be needed by applications that use OES. Factors that influence the qualities of service include, but are not limited to guarantees concerning:

1. generation of a notification event;
2. receipt of a notification event; or,
3. delivery of a notification event.

We presume that incoming requests to change a node's value are queued up for processing. For nodes which have values that frequently change, such as those used in highly interactive applications, it may be desirable to discard intermediate requests to change the values and silently drop the notification events generated by such actions, particularly when the system under heavy loads. Quite often, it is sufficient to simply process the most recently queued request, and silently drop earlier requests that had been queued, but not yet processed. In fact, this is prescribed to be the default behaviour; forced generation of a notification event for every operation performed on a node is offered, however, as an extended quality-of-service parameter.

We presume that clients taking advantage of OES exist in a partially disconnected environment; that is, the communication link between an OES server and its clients may, from time to time, be broken, and what's more, a client may be unreachable at the time a notification event must be delivered to that client. In most circumstances, such a condition indicates extreme network failure, e.g., the process which made the subscription has died or "gone away." In these cases, it is desirable to have the contract expire immediately. In fact, we prescribe this to be the default behaviour: a subscription contract immediately and implicitly expires upon failure to deliver a notification for a subscription placed using that contract.

In some circumstances, however, such implicit expiration of subscription contracts is undesirable, and it is preferred to keep the contract "alive" until its stated expiration time. This is offered as an extended quality-of-service parameter for subscription contracts. Additionally, it may be critical that the client receive all notification events generated for a subscription placed using such a contract—even while the client was disconnected—provided the communication link between the client and the OES server is re-established before the normal expiration of the subscription contract. Such a quality-of-service level requires that notifications be cached for possible future receipt, and although expensive, is offered as an extended quality-of-service parameter.

(N.B.: The quality of service parameter described next has numerous security and implementation issues and although has not been struck from this record, it is highly suspect and likely will not appear in any implementation.)

Highly interactive real-time collaborative applications often must remain aware of the connectivity of participants. For example, an instant messaging application must provide appropriate feedback to the user in the event a message could not be delivered to a contact. To garner this awareness, users may optionally request delivery- and receipt-status information for the notification generated for an action. This status information details to which subscribers a notification event was:

- delivered and received: the subscribing process could be contacted;

- delivered, but not received: the subscribing process could not be contacted, but the process has requested the guaranteed delivery level of quality of service previously discussed; and,
- not delivered: the subscribing process could not be contacted, and the process did not request the guaranteed delivery level of quality of service.

Network Topology

(N.B.: The functionality herein described will not be included in the first revision of the service.)

It is expected that the OES service will provided in a distributed fashion, with multiple machines responsible for parts of the tree. A distributed, (partially) replicated architecture, with a federation of servers maintaining consistency across machines will provide the greatest flexibility for scaling the architecture to higher and higher loads.

In particular, as we anticipate the use of the tree for sharing state information in highly interactive collaborative applications, we expect that some areas of the tree will experience extensive access. To prevent placing a bottleneck at OES server, we suppose that parts of the tree may be offloaded to one of the machines that are accessing it quite frequently. The central OES server maintains a shortcut to the machine that has leased responsibility for it, redirecting, for a period of time, requests for this part of the tree to the leaser. The leaser and the central server pass messages from time to time to ensure that a relatively recent copy of the sub-tree is available on the server, should the leaser suddenly “go away.” In effect, we propose that the communication topology be allowed to dynamically wander from a strict client-server model, to a more relaxed peer-to-peer model, as deemed appropriate by the application programmer.

Implementation

Wire Protocol

The wire protocol to be used in OES follows the conventions laid out by popular Internet protocols, such as HTTP [Fielding97], RVP [Osborne00], and DAV [Whitehead99]. This wire protocol follows a strict client-server interaction pattern in which the client always initiates a request, and the server always generates a reply. Except in the case of notification delivery, the term *client* always refers to application software that utilizes the OES service, and the term *server* always refers to the centralized process that maintains the tree and arbitrates access to it. During notification delivery, the client and server roles are swapped in the sense that the machine that runs the OES service initiates communication with the machines that running application software placing subscriptions.

Communication is done using TCP/IP stream sockets, or some similar guaranteed-delivery, stream-based sockets protocol (e.g., IrSock). Communication is conducted using UTF-7 encoded character streams. Lines of text are exchanged. Each line is delimited by a carriage-return (character code: U+000D)/linefeed (character code: U+000A) pair. Binary data, if it is to be transmitted, is encoded using the base64-encoding scheme.

After a connection between client and server is established, any number of request-reply exchanges may occur. A request-reply exchange follows a strict pattern, and begins with the request. In an augmented Backus-Naur Form (BNF) similar to that used by RFC 2068, an exchange is defined as:

```
OES-Exchange =          OES-Request OES-Reply
OES-Request =           OES-Request-Line
                        *message-header
                        CRLF
                        [ message-body ]
OES-Request-Line =      Verb SP [ Verb-URI SP ] “OES/1.0” CRLF
Verb =                  “LIST” | “GET” | “POST” | “REMOVE” | “NOTIFY” |
                        “LEASECONTRACT” | “CANCELCONTRACT” | “SUBSCRIBE” |
                        “UNSUBSCRIBE” |
OES-Version =           “OES/0.1”
verb-args =             *TEXT
OES-Reply =             OES-Reply-Line
                        *message-header
                        CRLF
                        [ message-body ]
OES-Reply-Line =        “OES/1.0” SP Status-Code SP Reason-Phrase CRLF
Status-Code =             “200” ; OK
                        | “400” ; Bad request
                        | “404” ; Not found
Reason-Phrase =         “OK” | “Bad request” | “Not found”
message-header =        field-name “:” field-value CRLF
field-name =            token
field-value =           *TEXT
message-body =          XML
```

As indicated, the message bodies consist exclusively of XML data. The schema that applies to XML exchanged is:

```
<? xml version="1.0" >
<Schema xmlns="urn:schemas-microsoft-com:xml-data" xmlns:dt="urn:schemas-
  microsoft-com:datatypes" >
```

```
<ElementType name='node' content='eltOnly' model='closed' order='one' >
   <AttributeType name='nodename' dt:type='string' required='yes' />
   <AttributeType name='nodepath' dt:type='string' required='no' />
   <AttributeType name='upl' dt:type='int' required='no' default='-1' />
   <AttributeType name='guaranteed' dt:type='boolean' required='no' default='0'
   />
   <AttributeType name='barricade' dt:type='boolean' required='no' default='0'
   />
   <AttributeType name='subscriptionPool' dt:type='string' required='no' />
   <attribute type='nodename' />
   <attribute type='upl' />
   <attribute type='guaranteed' />
   <attribute type='barricade' />
   <attribute type='subscriptionPool' />
   <attribute type='nodepath' />
   <ElementType name='content' content='textOnly' dt:type='bin.base64' />
   <element type='content' minOccurs='0' maxOccurs='1' />
   <element type='node' minOccurs='0' maxOccurs='*' />
</ElementType>
<ElementType name='notification' content='eltOnly' model='closed' order='one' >
   <ElementType name='lastValue' content='textOnly' dt:type='bin.base64' />
   <ElementType name='presentValue' content='textOnly' dt:type='bin.base64' />
   <element type='lastValue' minOccurs='0' maxOccurs='1' />
   <element type='presentValue' minOccurs='0' maxOccurs='1' />
</ElementType>
<ElementType name='contract content='eltOnly' model='closed' order='one'>
   <AttributeType name='contractid' dt:type='uuid' required='yes' />
   <AttributeType name='expiry' dt:type='dateTime' required='yes' />
   <AttributeType name='contactinfo' dt:type='string' required='yes' />
   <attribute type='contractid' />
   <attribute type='expiry' />
   <attribute type='contactinfo' />
   <ElementType name='subscription' content='empty' >
      <AttributeType name='nodepath' dt:type='string' required='yes' />
      <AttributeType name='depth' dt:type='int' required='yes' default='-1' />
      <AttributeType name='guaranteed' dt:type='boolean' required='no'
   default='0' />
      <attribute type='nodepath' />
      <attribute type='depth' />
      <attribute type='guaranteed' />
   </ElementType>
   <element type='subscription' minOccurs='0' maxOccurs='*' />
</ElementType>
</Schema>
```

Database Tables

NODE Table

This table stores the tree structure and the data associated with individual nodes. In essence, this table is the OES tree.

Field	**Type**	**Comment**
Nodeid	Autoid	Primary key
Parent_Nodeid	Integer	Parent node; constraint: 1=COUNT(SELECT * FROM NODE N WHERE N.Nodeid = [Parent_Nodeid])

Nodename	Char[64]	Name; constraint: 1=COUNT(SELECT * FROM NODE N WHERE N.Nodename = [Nodename])
Content	Char[10240]	Base64 encoded binary content
Upl	Integer	Upward propagation limit; constraint [Upl] >= -1; semantics: [Upl] = -1 means “infinite”; default: -1
QoS	Boolean	Guaranteed generation of notification for every operation; semantics: True (1) means “a notification is generated for every operation,” False (0) means “a notification might not be generated for a particular operation;” default: False (0)
Barricade	Boolean	Barricade is placed at this node; semantics: True (1) means “notifications will not propagate up the tree beyond this node, regardless of the value of the generating node’s upward propagation limit,” False (0) means “notifications will propagate up the tree beyond this node, contingent upon the generating node’s upward propagation limit;” default: False (0)
SubscriptionPool	Integer (Null allowed)	Subscription pool node; IF [SubscriptionPool] IS NOT NULL THEN 1=COUNT(SELECT * FROM NODE N WHERE N.Nodeid = [SubscriptionPool])

CONTRACT Table

This table stores the information needed to maintain subscription contracts.

Field	**Type**	**Comment**
Contractid	Autoid	Primary key
Contactinfo	String	Contact information; a string of the form “tcp:<host-ip-address>:<port>”
Expiry	Date/Time	Date and time this contract will expire

QoS	Boolean	Guaranteed delivery of subscription notifications for lifetime of contract; semantics: if True (1), then the contract persists until its stated date/time of expiration, even if a notification cannot be delivered; if False (0), the contract, and all its subscriptions, expire as soon as a notification cannot be delivered, or at the stated date/time of expiration, which ever is soonest

SUBSCRIPTION Table

This table stores the information needed to maintain individual subscriptions.

Field	**Type**	**Comment**
Contractid	Integer	Primary key; constraint: 1=COUNT(SELECT * FROM CONTRACT C WHERE C.Contractid = [Contractid])
Nodeid	Integer	Primary key; constraint: 1=COUNT(SELECT * FROM NODE N WHERE N.Nodeid = [Nodeid])
Depth	Integer	Depth of subtree to watch; constraint: [Depth] >= -1; sematics: [Depth] = -1 means "infinite;" default: -1
QoS	Boolean	Guaranteed delivery of most recent subscription notification; semantics: True (1) means "latest notification is cached for delivery when client unreachable," False (0) means "undeliverable notifications are discarded;" default: False (0)
LNPath	String (Null allowed)	Path of last node to generate a notification, when this notification must be cached for later delivery
LNOperation	Integer (Null allowed)	Numeric code to indicate operation that caused latest notification; sematics: 0 means "post," 1 means "touch," 2 means "remove"
LNContent	Char[10240]	Content associated with latest notification

Processing Logic

General Introduction

All verbs take, as an argument, a URI: this URI is the path to the node in the tree on which the verb operates. The client forms its request and sends it as a contiguous bundle to the server. The server, upon receiving the entire request, verifies that it conforms to the message standard described above. If the request is not a syntactically valid and complete request, status code 400 is returned. For this status code, there are no header lines, and no content lines returned. Otherwise, verb-specific processing, as indicated below, is performed.

LIST Verb

The LIST verb returns an enumeration of the names of all immediate children of the node specified as the argument URI. In this way, it is much like obtaining a directory listing of a file system folder. The XML tree returned has, as the document element, a <node> element for the node specified as the argument URI. For each immediate child of the specified node in the OES tree, a child <node> XML element appears in the XML returned. Each <node> element in the returned XML has at least the required nodename attribute present. It is generally presumed that <content> elements will not be sent.

The server, in response to a LIST request, parses the argument URI and locates the corresponding row in the NODE table. If no matching node is found, status code 402 is returned. For this status code, there are no header lines, and no content lines returned.

The response XML document is constructed by outputting a child <node> XML element for each row in the NODE table with a Parent_Nodeid that matches the Nodeid of the node specified by the argument URI.

Even if there are no child nodes enumerated, status code 200 is returned. The content returned is the ResponseXML produced by the pseudo-code above. The response includes at least the Content-Type and Content-Length message headers, and an XML document with at least one <node> element.

GET Verb

The GET verb returns the properties and content of OES tree node named in the argument URI. In this way, it is much like reading a file system file. The XML tree returned has, as the document element, a <node> element for the OES tree node specified in the argument URI. This document element will have at least all non-default valued attributes set, and will have a <content> element as a child—even if the content is empty.

The server, in response to a GET request, parses the argument URI and locates the corresponding row in the NODE table. If no matching node is found, status code 402 is returned. For this status code, there are no header lines, and no content lines returned.

Let ArgumentNode be the row in the NODE table corresponding to the argument URI. The following pseudo-code describes how the output XML is to be generated:

```
ResponseXML.createDocumentElement(“node”)
ResponseXML.documentElement.setAttribute(“nodename”,
ArgumentNode.Nodename)
ResponseXML.documentElement.setAttribute(“nodepath”,
ArgumentNode.Nodepath)
ResponseXML.documentElement.setAttribute(“upl”, ArgumentNode.Upl)
ResponseXML.documentElement.setAttribute(“guaranteed”, ArgumentNode.QoS)
ResponseXML.documentElement.setAttribute(“barricade”,
ArgumentNode.Barricade)
```

```
ResponseXML.documentElement.setAttribute(“subscriptionPool”,
ArgumentNode.subscriptionPool)
elem = ResponseXML.documentElement.createElement(“content”)
elem.text = ArgumentNode.Content
ResponseXML.documentElement.appendChild(elem)
```

The server returns status code 200. The content returned is the ResponseXML produced by the pseudo-code above. The response includes at least the Content-Type and Content-Length message headers.

POST Verb

The POST verb provides a facility for: the creation of new nodes in the OES tree; the update of an OES tree node’s contents and/or some or all of its attributes; and, “touching” an OES tree node, so as to trigger a notification, without actually changing the node’s contents or properties. The verb takes, as its argument URI, the path to the OES tree node to create or update. The message body of the request must be a XML document, with a single <node> element as the document element, which describes the attributes and content of the node that are to be set. This <node> document element can contain some or all of the non-required attributes; it may or may not contain a <content> child element.

If the OES tree node specified does not exist it is created, with any attributes not explicitly present in the request XML set to their default values. Similarly, if the <content> child element is present in the XML, it is used to initialize the content of the OES tree node. If the <content> child element is not present, the OES tree node is created without an initial content setting. If any of the node’s ancestors do not exist, they are also created: for each OES tree node created, its attributes are given their default values and it is created without an initial content setting. Status code 200 is returned; there are no message headers and no content returned for this request. Node creation *always* results in a notification being generated.

If the OES tree node specified already exists, the POST verb updates its attributes and content. Only those attributes explicitly present in the request XML are updated; if the <content> element is present in the request XML, then the content for the field is also updated. Status code 200 is returned; there are no message headers and no content returned for this request. Depending upon the quality of service specified for this node, changes to the node’s contents might or might not result in a notification being generated.

REMOVE Verb

The REMOVE verb provides a facility to remove individual nodes or entire sub-trees of the OES tree. The argument URI to this verb specifies the path to the root node of the OES sub-tree to remove. Thus, removing a particular node removes that node, its associated content, plus the descendant nodes and their contents. It is not possible to remove the root of the entire OES tree.

If the node specified by the argument URI does not exist, status code 404 is returned; otherwise, status code 200 is returned. There are no message headers and no content returned for this request. A notification is *always* generated for each node in the tree that was removed as a result of this operation.

LEASECONTRACT Verb

The LEASECONTRACT verb provides a facility for clients to take out a lease so that they may subscribe to notifications for nodes in the OES tree. The argument URI provides information on how to connect to the client, so that notifications may be delivered. This is typically a string of the form '`tcp:<ip-address>:<port>`' where <ip-address> and <port> are the numeric IP address and port number of the client's incoming notification receiving process.

This request has no content associated with it, but may have one or two message headers. The '`Quality-of-service`' header, if present, may take only the value '`keep-alive`.' If this header is specified, then the contract remains intact until its stated expiration time, even if a subscription notification could not be delivered before then. If this header is not specified, the default quality-of-service (contract expires as soon as a notification cannot be delivered) is in effect. The '`Contract-id`' header, if present, must specify a standing contract id returned as a successful response to a previous LEASECONTRACT request. If this header is specified, then the given lease is renewed and its expiration time is extended further into the future.

The reply to this verb is an XML document describing the contract leased or renewed. This XML document will have, as its document element, a <contract> element. All required attributes will be set for this element, but it will contain no child <subscription> elements.

CANCELCONTRACT Verb

The CANCELCONTRACT verb provides a facility for clients to prematurely revoke a contract they previously leased. Cancelling a contract immediately removes all subscriptions associated with the contract. The argument URI for this verb is a contract id returned in a previous successful LEASECONTRACT operation.

If the contract specified by the argument URI does not exist, status code 404 is returned; there are no message headers and no content returned for this status code. If the contract specified, however, could be found, then it is immediately cancelled. Status code 200 is returned; there are no message headers and no content returned for this status code.

SUBSCRIBE Verb

The SUBSCRIBE verb provides a facility for clients associate a subscription for notifications passing through a particular node in the OES tree with a contract they have leased. There can be only one subscription for a given node per contract, but many subscriptions (for many different nodes) may be placed with a given contract. At any rate, for a given notification-generating operation in the OES tree, only one notification will be delivered for a given contract, regardless of how many subscriptions placed with that contract may absorb the notification.

The SUBSCRIBE verb takes, as its argument URI, the path to the node to watch. There is no content associated with this request, but the request must include at least the '`Contract-id`' header: the value of this header must be a contract id returned as the result of a previous successful LEASECONTRACT request. This request may optionally specify a '`Cache-last-notification`' message header, which may take the value '`cache`' to indicate that the extended quality-of-service for subscription notification receipt is requested. The default behaviour, used when this message header is omitted, is for notifications that cannot be delivered to be silently dropped.

In response to a SUBSCRIBE request, the server validates that the node specified by the argument URI exists and that the contract specified in the `Contract-id` message header exists and has not yet expired. Either of these conditions is not met, status code 404 is returned; there are no message headers and no message body for this status code.

If the node and contract are found, the row for the subscription is added to the SUBSCRIPTION database if such a row does not already exist. Adding a second subscription for the same node under the same contract results in a modification to the SUBSCRIPTION database if the extended quality-of-service parameter changes; it does not generate an error. In either case, status code 200 is returned; there are no message headers and no message body for this status code.

UNSUBSCRIBE Verb

The UNSUBSCRIBE verb provides clients with a means to revoke a subscription previously placed with a SUBSCRIBE request. This verb takes, as its argument URI, the path to the node from which to unsubscribe. This request must specify the 'Contract-id' message header, the value of which must be the contract id specified in the previous SUBSCRIBE verb. There is no message body for this request.

In response to an UNSUBSCRIBE request, the server searches for a row in the SUBSCRIPTION table with the given node and contract id. If no matching row is found, status code 404 is returned; there are no message headers and no message body for this status code. Otherwise, if a matching row is found, the row is removed from the table and status code 200 is returned; there are no message headers and no message body for this status code. No further notifications will be processed for this subscription.

NOTIFY Verb

The NOTIFY verb is sent from the OES server to clients when a subscription notification is to be delivered. Though semantically not a request, the interchange between communicating processes follows the same request-response sequence all other interactions between OES servers and clients.

This verb takes, as its argument URI, the path of the node that generated the subscription notification. The message body contains an XML document describing the notification. This XML document has, as its document element, a <notify> element. This document element has its contract-id attribute set to the contract under which the subscription(s) that placed to capture this notification.

If the notification is being delivered as a result of a POST request, the XML document returned will have a <presentValue> child element. If the notification is being delivered as a result of a REMOVE request, the XML document returned will have a <lastValue> child element.

Future Considerations

In this section, we address some issues that will be considered in future revisions of this draft.

Security

The present specification does not make any provisions for important security considerations, such as authentication, access control, and wire privacy. Below, we elaborate how these concepts could be layered on top of the OES architecture just described.

Wire privacy and authentication can be integrated into transport-level security (TLS) features that rely on so-called public key infrastructure (PKI). The core idea here is that OES clients acting on behalf of some user present a PKI certificate to identify themselves; the server then issues a pass-phrase challenge to the client. The certificate and the pass-phrase are verified against an authentication database. A likely choice for an authentication database would be a directory service.

If the challenge fails, the connection is immediately aborted; requests from that client at that time are not processed. If the authentication challenge is successful, session certificates are exchanged: these certificates are used to encrypt requests and responses and thus provide privacy on the wire.

Access control is arguably best afforded using discretionary access control lists (DACLs) much along the same lines used in file-system security. The DACL is defined on the root of the sub-tree to which it pertains: as is the case in many file-systems, the security settings for an OES tree node is presumed to be inherited from ancestor nodes. The DACL names principles (i.e., users) or groups of principles permitted to perform an action. These actions could correspond to the verbs discussed in the previous section on the OES wire protocol. Before fulfilling a request, the server would verify that the authenticated principle has sufficient privilege. If the principle is not authorized, status code 403 (Forbidden) could be returned.

Authentication could be deferred until the DACL for a node indicates that authentication is required. In such a case, status code 401 (Unauthorized) could be returned. To support the subsequent authentication interaction phase, an AUTHENTICATE verb could be added to the wire protocol; this verb could then be issued by clients to start the authentication process.

We advocate that metadata such as DACLs be embedded into the tree as an accessible property on nodes and accessed in a fashion orthogonal to the way data is accessed. We hope that making the content bodies for all OES requests and replies XML documents will help in this area: DACLs could, for example, simply be transmitted a child element of the <node> document element of a GET reply or POST request.

Replication and Dynamic Partitioning

Replicating all or parts of the OES tree may be a technique amenable to improving service scalability, reliability, and performance. By replicating the tree across a farm of redundant servers, each equally capable of handling any request, and appropriately balancing the load across all machines in the farm, the service may scale to large numbers of clients. This is also a common technique for improving reliability.

However, as the size of the server farm increases, the overhead of synchronizing the tree across all machines becomes unbearable. Partitioning the tree so that responsibility for each main branch off the root (for example) is assigned to smaller, largely independent clusters of machines may help improve performance as the system scales.

Moreover, a general mechanism for dynamically partitioning the tree may also help improve responsiveness in synchronous real-time groupware applications. These applications typically exchange small chunks of data very frequently over short periods of time. Dynamic partitioning of the tree, where one of the individual groupware clients assumes responsibility for the small portion of the tree used to exchange groupware data (e.g., tele-pointer coordinates), could improve performance by relinquishing the server of the burden of maintaining this part of the tree. In much the same way a symbolic links are used in distributed file systems, shortcuts could be applied to nodes redirecting clients to another OES implementation responsible for the sub-tree rooted at that node. Clients, contacting the central server to access that part of the tree, could be returned status code 302 (Moved Temporarily) and a Location message header giving contact information for the client that has assumed responsibility for the sub-tree.

Responsibility for a sub-tree could be obtained on a timed lease, much like subscriptions. The central OES server implementation and the client that assumed responsibility for part of the sub-tree could, from time to time, synchronize data stores. Keeping a reasonably recent copy of the sub-tree on the server may help smooth fail-over if the client that assumed responsibility for the sub-tree were to unexpectedly "disappear." Full synchronization of course, should take as the lease expires.

Concurrency Control, Transactions and Sub-tree Locking

There are, however, no mechanisms present in the current design to afford concurrency control. It is presumed that operations are serialized at the server in some manner beyond the control of clients using this service. This, however, is insufficient when computation takes several nodes as input or outputs to several nodes. What is needed is a means to programmatically serialize access to nodes in the tree. Also closely related is the fact that our present design does not support transactional computations: those in which each sub-step must occur successfully and un-interleaved or else not at all.

To facilitate this, we propose a mechanism in which clients may obtain a temporary lock on a sub-tree of the hierarchy. While the lock is held, only those requests to access a node in the sub-tree that come from the lock holder are honoured. Other clients could have their requests denied and be returned status code 409 (Conflict) to indicate that the node requested was under a lock.

Locks are, of course, risky. They should operate on a timed basis, with the timeout reasonably short, and fairness algorithms should be used to prevent starvation when large numbers of clients are contending for access to the same sub-tree.

Expiration and No-Store Nodes

For some applications it is not important that the data at a node be persistently stored; for these applications a pure notification service is ideal. In earlier discussions about the design of this service, the notion of content expiration was pervasively used throughout the design. It was dropped in favour of the simpler design presented previously, but will be briefly mentioned here.

Expiration refers to a mechanism whereby nodes and their contents have an associated lifetime (expiry). Although the language used to describe expiry can be arbitrarily rich, we looked at the case where it can be expressed as a fixed point in time (e.g., now, in 50 minutes, never) or as a consequence of activity in the tree (e.g., when another value is stored at this node, when someone retrieves this value). When a node expires, the node is removed from the tree. When the content for a node expires, that content is disassociated from the node; the node itself, however, may continue to exist without content. (More on this shortly.)

Expiry opens up several nice behaviours not otherwise afforded by our design. A pure notification service is now possible by simply setting the content of a node to expire immediately: the posted content is communicated in any subscription notifications generated, but is immediately removed from the system as soon as the subscription notifications have been delivered.

With expiration, it is possible for a node to have multiple content values associated with it at a single instance in time: the most recently posted content not-yet-expired is assumed to be the "default value" or "top of the stack" and is the one returned when clients issue the GET verb to query for the node's value. However, as values expire, any previously posted not-yet-expired values take effect, in most-recently-posted-first order. This is useful in on-line presence awareness applications, for instance: one's status may have a default value "offline" that never expires, but a process started when one logs on to a workstation periodically sets and resets the value to "online" with, say, a one minute expiry. When one logs off and this process dies, the periodic refresh stops and within a minute one's status will revert to the default "offline" value.

Expiration applies uniformly to all concepts in OES. We have already described that subscription contracts are leased for specific periods of time, that is, have expiry. We suggest that DACLs, if implemented, could also have expiry. For example, imagine you are closing negotiations on the terms of the purchase of a new home. You want your realtor to have the easiest time to quickly get in touch with you today, without having to risk playing voice mail tag. You want your realtor to know when you are in your office and when you are at home because these are the places you are able to accept such calls. If this information was unobtrusively tracked and subsequently recorded in the OES tree—in itself, a substantial undertaking—you can grant your realtor the privilege to see this information for today only by applying an appropriate DACL entry that expires at the end of the day. The privilege is automatically revoked at the end of the day, when the DACL expires, without burdening you with the task of remembering to revoke the privilege manually and then actually doing it. By a similar argument, quality-of-service attributes could also have expiry.

Triggers and Server-Side Processing

In attentive user interfaces, many small changes are coalesced to make simpler, and in many cases, more meaningful inferences as to one's state. Consider a typical office: if the lights are on, and there is pressure on the seat of the chair, and the computer mouse is moving, then one could infer that there is someone seated in front of the computer, and thus might be accessible via computer-mediated communication channels. Each of these signals can be easily sensed, but alone or independent of one another they are not terribly useful for determining how best to reach someone. The problem is magnified if one considers hundreds of unique sensors, monitoring activity, some of which are changing very rapidly.

The useful information is the inference made out of the sensed inputs. We suppose that some sort of server-side processing that reacts to frequent changes in one part of the tree to provide slower-changing, higher-level semantic information out of these inputs. Clients could then subscribe to the higher-level semantic information and skip trying to gather all the intermediary inputs and make sophisticated inferences out of them.

Rather than leaving a dedicated client—an intelligent agent, if you will—running out there somewhere to listen to all these changes, we suppose that in a manner analogous to database triggers, arbitrary scripts could be bound to a node to provide a facility for this kind of processing on the server side. Of course, this entails numerous security and performance issues, some which could be addressed by DACLs and dynamic partitioning of the tree to push these inference computations out towards the edges of the network, closer to where the inputs are sensed.

References

[Fielding97]	Fielding, R., et al. IETF RFC 2068: Hypertext Transfer Protocol—HTTP/1.1. Publicly available from http://www.ietf.org/ and other RFC repositories.
[Osborne00]	Osborne, R., et al. RVP: A Presence and Instant Messaging Protocol. http://msdn.microsoft.com/library/techart/rvp.htm
[Whitehead99]	Whitehead, J. and Goland, Y. Y.. WebDAV: A network protocol for remote collaborative authoring on the Web. In Proceedings of the European Conference on Computer Supported Cooperative Work (ESCSW'99). http://www.ics.uci.edu/~ejw/papers/dav-ecscw.pdf