SIMPLE
Internet-Draft
Expires: December 22, 2003

J. Rosenberg
dynamicsoft

June 23, 2003

The Extensible Markup Language (XML) Configuration Access Protocol (XCAP)
draft-ietf-simple-xcap-00

Status of this Memo

Copyright Notice

Abstract

This specification defines the Extensible Markup Language (XML) Configuration Access Protocol (XCAP). XCAP allows a client to read, write and modify application configuration data, stored in XML format on a server. XCAP is not a new protocol. XCAP maps XML document sub-trees and element attributes to HTTP URIs, so that these components can be directly accessed by HTTP.

Table of Contents

1. Introduction

The Session Initiation Protocol for Instant Messaging and Presence Leveraging Extensions (SIMPLE) working group has been developing specifications for subscribing to, and receiving notifications of, user presence [10]. An important aspect of user presence is authorization policy. Indeed, the presence specification requires a Presence Agent (PA) to both authenticate and authorize all subscriptions before accepting them. However, it does not define how the server determines the authorization status of a subscriber. Users can set their authorization policy through web pages or voice response systems. However, there is currently no protocol specified for setting this policy. A protocol for this purpose is called an authorization manipulation protocol.

Mechanisms have also been defined to support reactive authorization [11][12]. Reactive authorization allows the user to be informed when someone has attempted to subscribe to their presence when the server is unable to determine an authorization policy. The user can then go and set an authorization policy for the subscriber, using the same unspecified mechanism for setting the policy.

Another important aspect of presence systems is the buddy list, also known as the presence list. This is a list of users that a watcher wishes to learn presence state for. This list can be stored in the client, or it can be stored in a centralized server. In the latter case, the client would subscribe to the list as a whole [13]. The presence list can be set by using a web page or voice response application. However, there is no protocol mechanism currently specified to manage the presence list. Such a protocol is called a presence list manipulation protocol.

The SIMPLE group has defined requirements for an authorization manipulation protocol and a presence list manipulation protocol. These protocols have similar requirements, and are captured in [14].

This document proposes a candidate for the authorization and presence manipulation protocol, called the Extensible Markup Language (XML) Configuration Access Protocol (XCAP). XCAP is not actually a new protocol. XCAP is a set of conventions for using HTTP to read, write and modify XML configuration data. XCAP is based heavily on ideas borrowed from the Application Configuration Access Protocol (ACAP) [15], but it is not an extension of it, nor does it have any dependencies on it. Like ACAP, XCAP is meant to support the configuration needs for a multiplicity of applications, rather than just a single one.

2. Overview of Operation

XCAP supports the needs of any application that needs access to data defined by clients of the application. Each application that makes use of XCAP specifies an application usage (Section 4). This application usage defines the XML schema [1] for the data used by the application, along with other key pieces of information. The principal task of XCAP is to allow clients to read, write, modify, create and delete pieces of that data. These operations are supported using HTTP 1.1 [2]. An XCAP server acts as a repository for collections of XML documents. There will be documents stored for each application. Within each application, there are documents stored for each user. Each user can have a multiplicity of documents for a particular application. To access some component of one of those documents, XCAP defines an algorithm for constructing a URI that can be used to reference that component. Components refer to any subtree of the document, or any attribute for any element within the document. Thus, the HTTP URIs used by XCAP point to pieces of information that are finer grained than the XML document itself.

With a standardized naming convention for components of XML documents, the basic operations for accessing the data are simple. Reading one of the components is just a standard HTTP GET operation. Writing, creating or modifying one of the components is a standard HTTP POST or PUT operation. Deleting a component is just a standard DELETE operation. For example, to add a friend to a presence list, a client would construct an XML document fragment which contains the information on that friend. The client would then construct a URI that refers to the location in the presence list document where this new fragment is to be added. The client then performs a POST operation against the URI, placing the document fragment into the body of the POST request. To provide atomic read/modify/write operations, the HTTP If-Unmodified-Since header field is used. The HTTP POST operation used by the client would contain the date obtained in the Last-Modified header field from the GET used to read the data.

3. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in RFC 2119 [3] and indicate requirement levels for compliant implementations.

4. Application Usages

A central concept in XCAP is that of an application usage. An application usage defines the way in which a specific application makes use of XCAP. This definition is composed of several pieces of information, such as an XML schema and constraints on values of one element given values in another.

Application usages are documented in specifications which convey this information. In particular, an application usage specification MUST provide the following information:

Application Usage ID (AUID): Each application usage is associated with a name, called an AUID. This name uniquely identifies the application usage, and is different from all other AUIDs. AUIDs exist in one of two namespaces. The first namespace is the IETF namespace. This namespace contains a set of tokens, each of which is registered with IANA. These registrations occur with the publication of standards track RFCs [16] based on the guidelines in Section 10. The second namespace is the vendor-proprietary namespace. Each AUID in that namespace is prefixed with the token "vnd", followed by a period ("."), followed by a valid DNS name, followed by another period, followed by any vendor defined token. A vendor creating such an AUID MUST only create one using domain names for which it is an administrator. As an example, the example.com domain can create an AUID with the value "vnd.example.com.foo" but cannot create one with the value "vnd.example.org.bar". AUIDs within the vendor namespace do not need to be registered with IANA. The vendor namespace is also meant to be used in lab environments where no central registry is needed.

MIME Type: Each application usage MUST register a MIME type for its XML documents. This is done based on the procedures of RFC 3023 [4].

XML Schema: Each application will have a unique schema which defines the data needed by the application. In XCAP, this schema is represented using XML schema. As an example, presence list data is composed of a list of URIs, each of which represends a member of presence list. [17] defines the XML schema for this data.

Additional Constraints: XML schemas can represent a variety of constraints about data, such as ranges and types. However, schemas cannot cover all types of data constraints, including constraints introduced by data interdependencies. For example, one XML element may contain an integer which defines the maximum number of instances of another element. The application usage defines these

additional constraints.

Data Semantics: The application usage needs to define detailed semantics for each piece of data in the schema.

Naming Conventions: The data defined by the XML schema will be used by any number of entities participating in the application. In the case of presence list, the data is used by the Resource List Server (RLS), which reads the data, and by the clients, which write it. During the execution of the application (i.e., the processing of the list subscription), specific documents will need to be read or written. In order for the application to function properly, there needs to be agreement on exactly which documents are read or written by the application. This is an issue of naming conventions; agreeing on how an application constructs the URI representing the document that is to be read or written. The application usage spells out this information.

Computed Data: Frequently, some of the data defined in the schema is not independent; that is, its value depends on the values of other elements in the document. As a result, when a client uses XCAP to modify the independent pieces of the document, the server needs to compute the dependent ones in order to fully populate the document. The application usage needs to define which data components are dependent, and how they are computed. As an example, when the URI for a presence list is not specified by a client, a URI is chosen by the server and filled in. This needs to be specified by the application usage.

Authorization Policies: By default, an XCAP server will only allow a user to access (read, write, delete or modify) their own documents. The application usage can specify differing default authorization policies. Of course, the default can always be overriden by operator or user-specified policies.

Application usages are similar to dataset classes in ACAP.

5. URI Construction

In order to manipulate a piece of configuration data, the data must be represented by an HTTP URI. XCAP defines a specific naming convention for constructing these URIs. This convention is very similar to the naming conventions used for dataset classes in ACAP, and makes use the XPath [5] specification for identifying nodes of an XML document.

The HTTP URI consists of two parts:

```
XCAP-URI        = Document-URI ["?" Node-Selector]
Document-URI    = http_URL      ;from RFC2616
Node-Selector   = *uric         ;Escape coded LocationPath from XPath
```

The first part, the Document-URI, selects a specific XML document. It is a valid HTTP URL, subject to the constraints described here. The constraints for constructing this URI are discussed below in Section 5.1. Once a document is selected, the remainder of the URI (the Node-Selector) identifies which components of the document are being addressed. The Node-Selector is an XPath [5] LocationPath expression, subject to constraints described below.

5.1 Identifying the XML Document

XCAP mandates that a server organizes documents according to a defined hierarchy. The root of this hierarchy is an HTTP URI called the XCAP services root URI. This URI identifies the root of the tree within the domain where all XCAP documents are stored. It can be any valid HTTP URL, but MUST NOT contain a query string. As an example, http://xcap.example.com/services might be used as the XCAP services root URI within the example.com domain. Typically, the XCAP services root URI is provisioned into client devices for bootstrapping purposes.

Beneath the XCAP services root URI is a tree structure for organizing documents. The first level of this tree consists of the XCAP AUID. So, continuing the example above, all of the documents used by the presence list application would be under http://xcap.example.com/services/presence-lists.

It is assumed that each application will have data that is set by users, and/or it will have global data that applies to all users. As a result, within the directory structure for each application usage, there are two sub-trees. One, called "users", holds the documents that are applicable to only specific users, and the other, called "global", holds documents applicable to all users.

Within the "users" tree are zero or more sub-trees, each of which identifies a documents that apply to a specific user. XCAP does not itself define what it means for documents to "apply" to a user, beyond specification of a baseline authorization policy. Specifically, the default authorization policy is that only a user who authenticates themself as user X can read, write, or otherwise access in any way the documents within sub-tree X. Each application usage can specify additional authorization policies which depend on data used by the application itself.

The remainder of the URI (the path following "global" or the specific user) is not constrained by this specification. The application usage MAY introduce constraints, or may allow any structure to be used.

5.2 Identifying the XML Nodes

The second component of the XCAP URI specifies specific nodes of the XML document which are to be accessed. Nodes, in this context, refers to the definition provided in the XPath specification, and therefore includes XML elements, attributes, text, namespaces, processing instructions, comments, and roots. These nodes are identified by a LocationPath expression, as defined in XPath. Either the abbreviated or unabbreviated form MAY be used.

Contraints are imposed on the XPath expression based on the operation being performed. These do not constrain the functions or axes that can be used in the XPath expression, but rather constrain the resulting node set. See Section 6 for details.

6. Client Operations

An XCAP client is an HTTP 1.1 compliant client. An XCAP client performs a set of primitive operations by invoking specific methods against the server, using specific URIs, where the requests contain bodies and headers subject to specific constraints. The set of primitive operations, the methods used to accomplish them, and the header and body constraints are described here.

6.1 Creating a New Document

To create a new document, the client constructs a URI that references the location where the document is to be placed. This URI MUST NOT contain a NodeSelector component, and MUST meet the constraints described in Section 5.1. The client then invokes a PUT method on that URI.

The content in the request MUST be an XML document compliant to the schema associated with the application usage defined by the URI. For example, if the client performs a PUT operation to http://xcap.example.com/services/presence-lists/users/joe/mybuddies, presence-lists is the application unique ID, and the schema defined by it would dictate the body of the request.

6.2 Replace an Existing Document

To replace an existing document with a new one, the procedures of Section 6.1 are followed; the Request-URI merely refers to an existing document which is to be replaced with the content of the request.

6.3 Deleting a Document

To delete a document, the client constructs a URI that references the document to be deleted. By definition this URI will not contain a NodeSelector component. The client then invokes a DELETE operation on the URI to delete the document.

6.4 Fetching a Document

As one would expect, fetching a document is trivially accomplished by performing an HTTP GET request with the Request URI set to the document to be fetched. It is useful for clients to perform conditional GETs using the If-Modified-Since header field, in order to check if a locally cached copy of the document is still valid.

6.5 Creating a New Element

To create a new XML element within an existing document, the client constructs a URI whose Document-URI points to the document to be modified. The Node-Selector MUST be present, containing an expression identifying the point in the document where the new element is to be added. The node-set selected by the expression MUST contain only a single XML element.

The client then invokes the HTTP POST method. The content in the request MUST be an XML document. That XML document MUST be conformant to the schema associated with the application usage defined by the URI. The server will insert the document such that the first element of the document becomes the next sibling immediately following the element specified by the Request-URI. The client SHOULD be certain, before making the request, that the resulting modified document will also be conformant to the schema.

6.6 Replacing an Element in the Document

Replacing an element of the document constitutes storage of a supplied entity under the specified URI, and is therefore accomplished with the PUT method (as opposed to POST, which will insert). The client constructs a URI whose Document-URI points to the document to be modified. The Node-Selector MUST be present, containing an expression identifying the element whose value is to be replaced. The node-set selected by the expression MUST contain only a single XML element.

The client then invokes the PUT method. The entity of the request MUST be of type text/plain. The server will take the value of the element specified by the request URI, and replace it with the content of the PUT request. Here, value refers to the binary contents of an XML document, starting with the beginning tag of the element, and ending with the end tag. This differs from the "string value" defined in XPath, which refers only to the values of the text element descendants of an element. The client SHOULD be certain, before making the request, that the resulting modified document will be conformant to the schema.

The body of the request here is of type text/plain because the value of an element need not be a valid XML document; frequently, it will be text or CDATA. Of course, the value of an XML element may be other XML elements, in which case the body of the request will be an XML document fragment, and by itself not compliant to any schema.

Note that this operation only modifies the value of an element. It cannot modify the attributes of the element. To do that, the replace attribute operation is performed.

6.7 Delete an Element

To delete elements from a document, the client constructs a URI whose Document-URI points to the document containing the elements to be deleted. The Node-Selector MUST be present, containing an expression identifying the elements to be deleted. Unlike most of the other operations, the node-set selected by the expression MAY contain multiple elements.

The client then invokes the HTTP DELETE method. All of the elements specified by the node set will be deleted by the server. The body of the request SHOULD be empty. The client SHOULD be certain, before making the request, that the resulting modified document will also be conformant to the schema.

6.8 Fetch an Element

To fetch an element of a document, the client constructs a URI whose Document-URI points to the document containing the element to be fetched. The Node-Selector MUST be present, containing an expression identifying the element whose value is to be fetched. The node-set selected by the expression MUST contain only a single XML element.

The client then invokes the GET method. The response will contain an XML document with the specified element as the one and only child of the document root.

> OPEN ISSUE: This only allows you to get one element at a time. We could allow the XPath expression to specify multiple elements, and then the response contains a document with each of those elements as a child of the document root. However, that document might not be compliant to the schema, and worse, the document doesnt actually reflect any specific sub-tree of the actual document.

6.9 Create an Attribute

To create an attribute in an existing element of a document, the client constructs a URI whose Document-URI points to the document to be modified. The Node-Selector MUST be present, containing an expression identifying an attribute that is to created. Specifically, the last location step of the expression MUST specify an attribute axis, and the context MUST specify a single element that exists within the document.

The client then invokes the HTTP POST method. The content defined by the request MUST be of type text/plain. A new attribute is added to the element defined by the context, with the name specified by the

node test in the last location step, with a value specified by the body of the request. If an attribute of this name already exists, it is replaced. The client SHOULD be certain, before making the request, that the resulting modified document will also be conformant to the schema.

6.10 Replacing Attributes

To replace an attribute in an existing element of a document, the client constructs a URI whose Document-URI points to the document to be modified. The Node-Selector MUST be present, containing an expression identifying an attribute that is to be replaced.

The client then invokes the HTTP PUT method. The content defined by the request MUST be of type text/plain. The value of the attribute defined by the Node-Selector is replaced by the body of the request. The client SHOULD be certain, before making the request, that the resulting modified document will also be conformant to the schema.

6.11 Deleting Attributes

To delete attributes from the document, the client constructs a URI whose Document-URI points to the document containing the attributes to be deleted. The Node-Selector MUST be present, containing an expression identifying the attributes to be deleted. Unlike most of the other operations, the node-set selected by the expression MAY contain multiple attributes.

The client then invokes the HTTP DELETE method. All of the attributes specified by the node set will be deleted by the server. The body of the request SHOULD be empty. The client SHOULD be certain, before making the request, that the resulting modified document will also be conformant to the schema.

6.12 Fetching Attributes

To fetch an attribute of a document, the client constructs a URI whose Document-URI points to the document containing the attribute to be fetched. The Node-Selector MUST be present, containing an expression identifying the attribute whose value is to be fetched. The node-set selected by the expression MUST contain only a single XML attribute.

The client then invokes the GET method. The response will contain an text/plain document with the value of the specified attribute.

6.13 Fetching Metadata

Elements and attributes in an XML document have various meta-data associated with them. For example, and XML element has a certain number of child elements. That number is a piece of meta-data that describes the element. Currently, there is no way to fetch meta-data for XML elements or attributes.

OPEN ISSUE: Is this restriction OK? XPath does specify functions for computing meta-data about node sets. We can't use them since XCAP mandates that the request URI be a location set, which does not include these other functions. We could relax the constraint if this is deemed important.

6.14 Read/Modify/Write Transactions

It is anticipated that a common operation will be to read the current version of a document or element, modify it on the client, and then write the change back to the server. In order for the results to be consistent with the client's expectations, the operation must be atomic.

To accomplish this, the client stores the value of the Last-Modified header field from the response to its GET operation used to read the element, attribute, or document that is to be modified. To guarantee atomicity, the PUT or POST operation used to write the changes back to the server MUST contain an If-Unmodified-Since header field, whose value is equal to the value from the prior GET response. If the request fails with a 412 response, the client knows that another update of the data has occurred before it was able to write the results back. The client can then fetch the most recent version, and attempt its modification again.

7. Server Behavior

TODO: Specify an XML body type for the responses that contains error conditions or success results.

An XCAP server is an HTTP 1.1 compliant origin server. The behaviors mandated by this specification relate to the way in which the HTTP URI is interpreted and the content is constructed.

An XCAP server MUST be explicitly aware of the application usage against which requests are being made. That is, the server must be explicitly configured to handle URIs for each specific application usage, and must be aware of the constraints imposed by that application usage.

OPEN ISSUE: It may be possible to remove this restriction by allowing an application usage to define operation in a server that doesnt understand the usage. That may require some capabilities discovery to be introduced, this constraint didnt seem that problematic.

When the server receives a request, the treatment depends on the URI. If the URI refers to an application usage not understood by the server, the server MUST reject the request with a 404 (Not Found) response. If the URI refers to a user that is not recognized by the server, it MUST reject the request with a 404 (Not Found).

Next, the server authenticates the request. All XCAP servers MUST support HTTP Digest [6]. Furthermore, servers MUST support HTTP over TLS, RFC 2818 [7]. It is RECOMMENDED that administrators use an HTTPS URI as the XCAP root services URI, so that the digest client authentication occurs over TLS.

Next, the server determines if the client has authorization to perform the requested operation on the resource. The default authorization policy is that only client X can access (create, read, write, modify or delete) resources under the "users/X" directory. An application usage can specify an alternate default authorization policy specific to that usage. Of course, an administrator or privileged user can override the default authorization policy, although this specification provides no means for doing that. Generally, if users need to be able to control authorization for access to XCAP data, an XCAP application usage should be specified which allows the user to set the policies as needed.

OPEN ISSUE: This is different from ACAP, where authorization policies are built into the protocol. I think the default generally will suffice, so I would rather not burden the baseline

protocol with it.

Once authorized, the specific behavior depends on the method and what the URI refers to.

7.1 POST Handling

If the URI contains only a Document-URI, the server examines the entity body of the request. If there is no entity in the body, the server MUST reject the request with a 409 response. If there is an entity, but it is not well-formed, the server MUST reject the request with a 409 response. If it is well-formed, but not compliant to the schema associated with the application usage, the server MUST reject the request with a 409 response. If it is compliant to the schema, the server MUST store the document at the requested URI. If there is not already a document stored at that URI, a 201 (Created) response code MUST be sent, and it MUST include a Location header field containing the value of the URI for the document (which will be the same as the one in the Request-URI). Otherwise, a 200 OK response is returned, and the document in the body replaces the existing one at that URI. For either a 200 or 201 response, the new document is returned in the body of the response, with a Content-Type equal to the MIME type defined by the application usage.

If the Request URI contains a Node-Selector, the server MUST verify that the document defined by the Document-URI exists. If no such document exists on the server, the server MUST reject the request with a 409 response code. If the document does exist, the server evaluates the Node-Selector as an XPath RelativeLocationPath, relative to the root of the document. If the Node-Selector does not comply to the grammar for RelativeLocationPath, the server MUST reject the request with a 400 response code. If the Node-Selector does comply, and it evaluates to anything other than the empty set, a single attribute node or single element node, the server MUST reject the request with a 409 response code.

If the Node-Selector evaluates to the empty set, and the last location step is on the attribute axis, and the expression without the last location step evaluates to a single element node, the server adds an attribute to that element. Its name is the name given in the node test of the last location step, and its value is taken from the body of the request. The server then generates a 200 OK response, whose body contains the value of the attribute, with a Content-Type of text/plain.

If the Node-Selector evaluates to a single element node, the server takes the content of the request, and inserts it into the document specified by the URI such that the selected element is the immediate

sibling of the nodes defined by the content of the request. The server then generates a 200 OK response, whose body contains the parent element of the new elements just inserted. The parent element is represented by extracting the contents of the XML document, starting with, and including, the begin tag of the element, up to, and including, the end tag for the element. The Content-Type of the response is set to application/xml.

OPEN ISSUE: We can't use the MIME type for the application usage, since the schema may not allow for the document to start with any element defined by the schema. Is that OK? I think so.

If the Node-Selector evaluates to a single attribute node, the server takes the content of the request, and sets it as the value of the attribute specified by the body of the request. The server then generates a 200 OK response, whose body contains the value of the attribute, with a Content-Type of text/plain.

If the result of the POST is a document which does not comply with the XML schema for the application usage, the server MUST NOT perform the POST, and MUST reject the request with a 409 (Conflict).

7.2 PUT Handling

When the Request URI contains only the Document-URI, the semantics of PUT are as defined in HTTP 1.1 Section 9.6 - the content of the request is placed at the specified location.

If the Request URI contains a Node-Selector, the server MUST verify that the document defined by the Document-URI exists. If no such document exists on the server, the server MUST reject the request with a 409 response code. If the document does exist, the server evaluates the Node-Selector as an XPath RelativeLocationPath, relative to the root of the document. If the Node-Selector does not comply to the grammar for RelativeLocationPath, the server MUST reject the request with a 400 response code. If the Node-Selector does comply, and it evaluates to anything other than the a single element node or attribute node, the server MUST reject the request with a 409 response code.

If the Node-Selector evaluates to a single element node, the server takes the content of the request, and replaces the value of that element (where value is defined as all of the content - elements, text, or CDATA - between the begin and end tags of the element) with that content. The server then returns a 200 OK response.

OPEN ISSUE: PUT is not quite right here, since a subsequent GET on the same URI will not return exactly the same thing - the begin

and end tags will be present. This may need to be POST, but then, how to differentiate a replace with an append operation?

If the Node-Selector evaluates to a single attribute node, the server takes the content of the request, and sets it as the value of the attribute. It then returns a 200 OK response.

If the result of the PUT is a document which does not comply with the XML schema for the application usage, the server MUST NOT perform the PUT, and MUST reject the request with a 409 (Conflict).

7.3 GET Handling

If the request URI contains only a Document-URI, the server returns the document specified by the URI if it exists, else returns a 404 response.

If the request URI specifies a Node-Selector, the server verifies that the document specified by the Document-URI exists. If it does not exist, the server returns a 404 (Not Found) response. If the document does exist, the server evaluates the Node-Selector as an XPath RelativeLocationPath, relative to the root of the document. If the Node-Selector does not comply to the grammar for RelativeLocationPath, the server MUST reject the request with a 400 response code. If the Node-Selector does comply, and it evaluates to anything other than the a single element node or attribute node, the server MUST reject the request with a 409 response code.

If the Node-Selector evaluates to a single element node, the server takes the document text, starting with, and including, the begin tag of the element, up to, and including, the end tag for the element, and places it into the body of a 200 OK response, setting the Content-Type to application/xml.

If the Node-Selector evaluates to a single attribute node, the server takes the value of the attribute and returns it as the content of the 200 OK response, setting the Content-Type to text/plain.

OPEN ISSUE: Do we need to say anything about HEAD? We havent said anything about meta-data so far; most of that is just regular HTTP usage, I think.

7.4 DELETE Handling

If the request URI contains only a Document-URI, the server deletes the document specified by the URI if it exists and returns a 200 OK response, else returns a 404 response.

If the request URI specifies a Node-Selector, the server verifies that the document specified by the Document-URI exists. If it does not exist, the server returns a 404 (Not Found) response. If the document does exist, the server evaluates the Node-Selector as an XPath RelativeLocationPath, relative to the root of the document. If the Node-Selector does not comply to the grammar for RelativeLocationPath, the server MUST reject the request with a 400 response code. If the Node-Selector does comply, and it evaluates to the empty set, the server MUST reject the request with a 404 (Not Found).

Otherwise, the server removes all of the data defined by the node-set. Specifically, any elements in the node set are removed from the document, and any attributes in the node set are removed from the document. It then returns a 200 OK response.

If the result of the deletion is a document which does not comply with the XML schema for the application usage, the server MUST NOT perform the deletion, and MUST reject the request with a 409 (Conflict).

7.5 Managing Modification Times

An XCAP server MUST maintain modification times for all resources that can be referenced by a URI. Specifically, this means that each document, and within the document, each element and attribute, MUST be associated with a modification time maintained by the server. These modification times are needed to support condition GET, POST and PUT requests.

When a PUT or POST request is made that creates or replaces a document, the modification time of that document and all elements and attributes within is set to the current time. When a PUT request is made to a URI referencing an XML element, the modification time of that element and all of its enclosed children and their attributes is set to the current time. Furthermore, the modification time of all elements which are ancestors of that element have their modification time set to the current time. However, the modification times of attributes belong to elements that are ancestors of the modified element do not have their modification times changed.

When a POST request is made to a URI referencing an XML element, the modification time of all of the elements and their attributes within the document in the body of the request is set to the current time. Furthermore, the modification time of the element which is the new parent of the elements in the request, and all of its ancestors, have their modification time set to the current time. However, the modification times of their attributes are unchanged.

When a POST request is made to a URI referencing an XML attribute, the modification time of that attribute, its element, and all elements that are ancestors of that element is set to the current time.

When a DELETE request is made to a URI referencing an element, the modification time of all ancestors of that element is set to the current time. When a DELETE request is made to a URI referencing an attribute, the modification time of its element, and all ancestors of that element, is set to the current time.

8. Examples

This section goes through several examples, making use of the presence-lists [17] XCAP application usage.

First, a user Bill creates a new presence-list, initially with no users in it:

```
PUT
http://xcap.example.com/services/presence-lists/users/bill/fr.xml HTTP/1.1
Content-Type:application/presence-lists+xml

<?xml version="1.0" encoding="UTF-8"?>
<presence-lists xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <list name="friends" uri="sip:friends@example.com" subscribable="true">
  </list>
</presence-lists>
```

Next, Bill adds an entry to the list:

```
PUT
http://xcap.example.com/services/presence-lists/users/bill/fr.xml?
presence-lists/list[@name="friends"] HTTP/1.1
Content-Type:text/plain

<entry name="Bill" uri="sip:bill@example.com">
  <display-name>Bill Doe</display-name>
</entry>
```

Note how the URI in the PUT request selects the list element whose name attribute is "friends". The body of that request replaces the existing value of that element, which was empty.

Next, Bill adds another entry to the list, which is another list that has three entries:

```
POST
http://xcap.example.com/services/presence-lists/users/bill/fr.xml?
presence-lists/list[@name="friends"]/entry[@name="Bill"] HTTP/1.1
Content-Type:text/plain

<list name="close-friends" uri="sip:close-friends@example.com"
      subscribable="true">
   <entry name="Joe" uri="sip:joe@example.com">
     <display-name>Joe Smith</display-name>
   </entry>
   <entry name="Nancy" uri="sip:nancy@example.com">
     <display-name>Nancy Gross</display-name>
   </entry>
   <entry name="Petri" uri="sip:petri@example.com">
     <display-name>Petri Aukia</display-name>
   </entry>
</list>
```

Then, Bill decides he doesnt want Petri on the list, so he deletes the entry:

```
DELETE
http://xcap.example.com/services/presence-lists/users/bill/fr.xml?
presence-lists/list/list/entry[@name="Petri"] HTTP/1.1
```

Bill decides to check on the URI for Nancy:

```
GET
http://xcap.example.com/services/presence-lists/users/bill/fr.xml?
presence-lists/list/list/entry[@name="Nancy"]/@uri HTTP/1.1
```

and the server responds:

```
HTTP/1.1 200 OK
Content-Type:text/plain

sip:nancy@example.com
```

9. Security Considerations

Frequently, the data manipulated by XCAP contains sensitive information. To avoid eavesdroppers from seeing this information, XCAP RECOMMENDS that an admistrator hand out an https URI as the XCAP root services URI. This will result in TLS-encrypted communications between the client and server, preventing any eavesdropping.

Client and server authentication are also important. A client needs to be sure it is talking to the server it believes it is contacting. Otherwise, it may be given false information, which can lead to denial of service attacks against a client. To prevent this, a client SHOULD attempt to upgrade [8] any connections to TLS. Similarly, authorization of read and write operations against the data is important, and this requires client authentication. As a result, a server SHOULD challenge a client using HTTP Digest [6] to establish its identity, and this SHOULD be done over a TLS connection.

10. IANA Considerations

This specification instructs IANA to create a new registry for XCAP application usage IDs (AUIDs).

XCAP AUIDs are registered by the IANA when they are published in standards track RFCs. The IANA Considerations section of the RFC must include the following information, which appears in the IANA registry along with the RFC number of the publication.

Name of the AUID. The name MAY be of any length, but SHOULD be no more than twenty characters long. The name MUST consist of alphanum [9] characters only.

Descriptive text that describes the application usage.

Normative References

[1] Thompson, H., Beech, D., Maloney, M. and N. Mendelsohn, "XML Schema Part 1: Structures", W3C REC REC-xmlschema-1-20010502, May 2001.

[2] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[3] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[4] Murata, M., St. Laurent, S. and D. Kohn, "XML Media Types", RFC 3023, January 2001.

[5] Clark, J. and S. DeRose, "XML Path Language (XPath) Version 1.0", W3C REC REC-xpath-19991116, November 1999.

[6] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A. and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[7] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[8] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", RFC 2817, May 2000.

[9] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

Informative References

[10] Rosenberg, J., "A Presence Event Package for the Session Initiation Protocol (SIP)", draft-ietf-simple-presence-10 (work in progress), January 2003.

[11] Rosenberg, J., "A Watcher Information Event Template-Package for the Session Initiation Protocol (SIP)", draft-ietf-simple-winfo-package-05 (work in progress), January 2003.

[12] Rosenberg, J., "An Extensible Markup Language (XML) Based Format for Watcher Information", draft-ietf-simple-winfo-format-04 (work in progress), January 2003.

[13] Rosenberg, J., Roach, A. and B. Campbell, "A Session Initiation Protocol (SIP) Event Notification Extension for Resource Lists", draft-ietf-simple-event-list-04 (work in progress), June 2003.

[14] Rosenberg, J. and M. Isomaki, "Requirements for Manipulation of Data Elements in Session Initiation Protocol (SIP) for Instant Messaging and Presence Leveraging Extensions (SIMPLE) Systems", draft-ietf-simple-data-req-02 (work in progress), April 2003.

[15] Newman, C. and J. Myers, "ACAP -- Application Configuration Access Protocol", RFC 2244, November 1997.

[16] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[17] Rosenberg, J., "An Extensible Markup Language (XML) Configuration Access Protocol (XCAP) Usage for Presence Lists", draft-rosenberg-simple-xcap-list-usage-00 (work in progress), May 2003.

Author's Address

Jonathan Rosenberg
dynamicsoft
600 Lanidex Plaza
Parsippany, NJ 07054
US

Phone: +1 973 952-5000
EMail: jdrosen@dynamicsoft.com
URI: http://www.jdrosen.net

Intellectual Property Statement

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.