SIMPLE
Internet-Draft
Expires: April 26, 2004

J. Rosenberg
dynamicsoft

October 27, 2003

The Extensible Markup Language (XML) Configuration Access Protocol (XCAP)
draft-ietf-simple-xcap-01

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

In many communications applications, such as Voice over IP, instant messaging, and presence, it is necessary for network servers to access per-user information in the process of servicing a request. This per-user information resides within the network, but is managed by the end user themselves. Its management can be done through a multiplicity of access points, including the web, a wireless handset, or a PC application.

Examples of per-user information are presence [12] authorization policy and presence lists. Presence lists are lists of users whose presence is desired by a watcher. Presence information for the list of users can be obtained by subscribing to a resource which represents that list [15]. In this case, the Resource List Server (RLS) requires access to this list in order to process a SIP [11]SUBSCRIBE [20] request for it. Requirements for manipulation of presence lists and authorization policies have been specified by the SIMPLE working group [16].

This specification describes a protocol that can be used to manipulate this per-user data. It is called the Extensible Markup Language (XML) Configuration Access Protocol (XCAP). XCAP is not a new protocol. Rather, it is a set of conventions for mapping XML documents and document components into HTTP URIs, rules for how the modification of one resource affects another, data validation constraints, and authorization policies associated with access to those resources. Because of this structure, normal HTTP primitives can be used to manipulate the data. XCAP is based heavily on ideas borrowed from the Application Configuration Access Protocol (ACAP) [18], 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

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 mapping components of XML documents to HTTP URIs, the basic operations for accessing the data are provided by existing HTTP primitives. Reading one of the components is accomplished with HTTP GET, creating or modifying one of the components is done with an HTTP PUT, and removing one of the components is done with an HTTP DELETE. To provide atomic read/ modify/write operations, HTTP entity tags are used.

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 [19] based on the guidelines in Section 10. The second namespace is the vendor-proprietary namespace. Each AUID in that namespace is prefixed with the reverse domain name name of the organization creating the AUID, followed by a period, followed by any vendor defined token. As an example, the example.com domain can create an AUID with the value "com.example.foo" but cannot create one with the value "org.example.foo". 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. The syntax for AUIDs, expressed in ABNF [7] (and using some of the BNF defined in RFC 2396 [8]) is:

```
AUID             =  global-auid / vendor-auid
global-auid      =  auid
auid             =  alphanum / mark
vendor-auid      =  rev-hostname "." auid
rev-hostname     =  toplabel *( "." domainlabel  )
domainlabel      =  alphanum
                    / alphanum *( alphanum / "-" ) alphanum
toplabel         =  ALPHA / ALPHA *( alphanum / "-" ) alphanum
```

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 [1].

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.

Resource Interdependencies: In many cases, when a user modifies an XCAP resource, many other resources need to change as well. Such interdependencies are application usage dependent. As an example, when a user performs a PUT operation to create a new presence list, the server may need to fill in the URI associated with that list. These interdependencies need to be specified by the application usage. Note that, if a server needs to modify data within a document just PUT by the client, this modification is effectively accomplished as a separate transaction. Concretely, this means that, after the server modifies the data, the entity tags are updated as if the client had made the change itself.

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. An application usage can also specify whether another application usage is used to define the authorization policies. An application usage for setting authorization policies can also be defined subsequent to the definition of the the main application usage. In such a case, the main application usage needs only to specify that such a usage will be defined in the future.

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. In particular, the host part identifies the XCAP server. The abs_path component of the HTTP URI identifies the specific XML document to be modified. XCAP servers organize XML documents in a specific hierarchical fashion, as described in Section 5.1. The URI MAY contain a query. This query is called a node selector. When present, it contains an XML component identifier formatted according to Section 5.2. The node selector identifies the specific component of the XML document. The HTTP URI without the query is called the document URI. , and makes use the specification for identifying nodes of an XML document.

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 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. A node refers to either an XML element or an attribute of an element. The node selector is an expression which identifies an element or attribute. Its grammar is:

```
node-selector          =  element-selector ["/" attribute-selector]
element-selector       = step *( "/" step)
step                   = by-name / by-pos / by-attr
by-name                = QName      ; from XML Namespaces
by-pos                 = QName "[" position "]"
position               = 1*DIGIT
by-attr                = QName "[" "@" att-name "=" <">
                           att-value <"> "]"
att-name               = QName
att-value              = AttValue   ; from XML specification
by-pos                 = QName "[" position "]"
position               = *DIGIT
attribute-selector     = "@" att-name
```

The node selector is based on the concepts in XPath [5]. Indeed, the node selector expression happens to be a valid XPath expression. However, XPath provides a set of functionality far richer than is needed here, and its breadth would introduce complexity into XCAP.

To determne the XML element or attribute selected by the node selector, processing begins at the root of the XML document. The first step in the element selector is then taken. Each step chooses a specific XML element within the current document context. The document context is the point within the XML document from which a specific step is evaluated. The document context begins at the root of the document. When a step determines an element within that context, that element becomes the new context for evaluation of the next step. Each step can select an element by its name, by a combination of name and attribute value, or by name and position. If

the step is attempting selection by name, the server looks for all elements within the current context with that name. Name matching is performed as described below. If there is more than one element with the specified name, the result is considered a no-match.

If the step is attempting selection by name and attribute, the server looks for all elements within the current document context with that name. Of those that match, it looks for ones that have the given attribute name, where that attribute has the given value. If there is no match, or if more than one element matches, the result is considered a no-match.

If the step is attempting selection by name and position, the server looks for all elements within the current document context with that name. These are then sorted in document order, as defined by Xpath. The position-th element is then selected. If there are fewer than position number of elements with that name, the result is considered a no-match.

Once the last step is executed, if there is no attribute selector, the result of the node selection is the last selected element. If there is an attribute selector, the server checks to see if there is an attribute with that name within the currently selectoed element. If there is not, the result is considered a no-match. Otherwise, that attribute is selected.

Matching of element names and attributes is performed by expanding them into the expanded name form, as described in XML Namespaces, and then performing the comparison of the results. When evaluating the QNames in the node selector, the default namespace and namespace definitions from the document URI apply.

As an example, consider the following XML document:

```
<?xml version="1.0"?>
   <watcherinfo xmlns="urn:ietf:params:xml:ns:watcherinfo"
                version="0" state="full">
     <watcher-list resource="sip:professor@example.net" package="presence">
       <watcher status="active"
                id="8ajksjda7s"
```

```
                duration-subscribed="509"
                event="approved" >sip:userA@example.net</watcher>
        <watcher status="pending"
                id="hh8juja87s997-ass7"
                display-name="Mr. Subscriber"
                event="subscribe">sip:userB@example.org</watcher>
      </watcher-list>
    </watcherinfo>
```

The node selector "watcherinfo/watcher-list/ watcher[@id="8ajksjda7s"]" would select the following XML element:


```
        <watcher status="active"
                id="8ajksjda7s"
                duration-subscribed="509"
                event="approved" >sip:userA@example.net</watcher>
```

6. Client Operations

An XCAP client is an HTTP 1.1 compliant client. Specific data manipulation tasks are accomplished by invoking the right set of HTTP methods with the right set of headers on the server. This section describes those in detail

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. 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-Match header field, in order to check if a locally cached copy of the document is still valid. An HTTP server MUST return Etags for entities that represent resources managed by XCAP.

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 in the URI. The node selector is constructed such that it meets two constraints. First, if evaluated against the current document, the result is a no-match. Secondly, if the element was added to the document as desired by the client, the node selector would select that element.

The client then invokes the HTTP PUT method [[OPEN ISSUE: what is the content type?]]. The content in the request MUST be an XML element. The server will insert the element into the document such that the node selector, if evaluated by the server, would return the content present in the request. The client SHOULD be certain, before making the request, that the resulting modified document will also be conformant to the schema.

It is important to note that the element might potentially be inserted in the document in several different ways, and still meet the constraints defined above. This is analagous to the case when a new file is PUT into a directory on a server; the location of that file within the directory is not specified, and is up the local file system to decide. The only guarantee is that GET(PUT(x)) returns document x.

6.6 Replacing an Element in the Document

Replacing an element of the document is also accomplished with PUT. The only difference with the behavior above for insertion is that the node selector, when evaluated against the current document, is a match for an element in the current document. That element is removed, and replaced with the content of the PUT request.

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, and identify the specific element to be deleted.

The client then invokes the HTTP DELETE method. The server will remove the element from the document. 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, and must identify the

element to be fetched.

The client then invokes the GET method. The response will contain that XML element. Specifically, it contains the content of the XML document, starting with the opening bracket for the begin tag for that element, and ending with the closing bracket for the end tag for that element.

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. The node selector is constructed such that it meets two constraints. First, if evaluated against the current document, the result is a no-match. Secondly, if the attribute was added to the document as desired by the client, the node selector would select that attribute.

The client then invokes the HTTP PUT method. The content defined by the request MUST be compliant to the grammar for Attribute as defined in XML 1.0 [[OPEN ISSUE: content type?]]. The server will add that attribute such that, if the node selector is evaluated on the resulting document, it returns the attribute present in the request. 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

Replacing an attribute of the document is also accomplished with PUT. The only difference with the behavior above for insertion is that the node selector, when evaluated against the current document, is a match for an attribute in the current document. That attribute is removed, and replaced with the content of the PUT request.

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, and evaluate to an attribute in the document to be deleted.

The client then invokes the HTTP DELETE method. The server will remove the attribute from the document. 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 client then invokes the GET method. The response will contain document with the specified attribute, formatted according to the grammar of Attribute as defined in the XML 1.0 specifications.

6.13 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 makes use of entity tags returned by the server in a GET operation used to read the element, attribute, or document that is to be modified. To guarantee atomicity, the PUT operation used to write the changes back to the server MUST contain an If-Match header field, whose value is equal to the entity tag 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.

Because there are no batching operations defined in HTTP, that would allow for a number of separate create, modify or delete operations to be performed atomically, designers of application usages should take care to structure their schemas so that operations that need to be performed atomically can be done in a single operation.

7. Server Behavior

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. Furthermore, an XCAP server MUST be aware of all of the XML namespaces present in any documents it manages. This is to ensure that any data constraints or data interdependencies imposed by a future application usage are properly supported by the server. It is also required to ensure that authorization policies are properly implemented.

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 [9]. 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. The server may also know of an application usage that itself defines authorization policies for another application usage. Of course, an administrator or privileged user can override the default authorization policy, although this specification provides no means for doing that.

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

7.1 POST Handling

Resources managed by XCAP do not represent processing scripts. As a

result, POST operations to XCAP URIs is not defined. A server receiving such a request SHOULD return a 405.

7.2 PUT Handling

The behavior of a server in receipt of a PUT request is as specified in HTTP 1.1 Section 9.6 - the content of the request is placed at the specified location. This section serves to define the notion of "placement" and "specified location" within the context of XCAP resources.

If the request URI represents a document (i.e., there is no node selector component), the content of the request MUST be a valid XML document, and MUST be compliant to the schema associated with the application usage in the URI. If it is not, the request MUST be rejected with a 409 response. If the request URI matches a document that exists on the server, that document is replaced by the content of the request. If the request URI does not match a document that exists on the server, the server adds the document to its repository, and associates it with the URI in the request URI. Note that this may require the creation of one or more "directories" on the server.

If the Request URI represents an XML element (i.e., it contains a node selector, but no attribute 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. The content of the request MUST be a single XML element and associated content (including children elements). If the request URI matches an element within the document, that element is removed, and replaced with the content of the request. If the request URI does not match an element in the document, the server inserts the content of the request as a new element in the document, such that the resulting document is compliant to the schema, and such that the request URI, when evaluated, would now point to the element which was inserted. There may be more than one way to perform such an insertion; in that case, it is the discretion of the implementor as to how it is done. It may also be possible that the insertion cannot be done without other additional elements being inserted, or cannot be done because the new element is not compliant to the schema. In such a case, the server MUST return a 409 response code. In all cases, the resulting document MUST be compliant to the schema.

If the Request URI represents an XML attribute (i.e., it contains a node selector and an attribute 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. The content of the request MUST be a single XML attribute, compliant to the grammar for Attribute as defined in XML 1.0 (i.e., name=value). If the request URI matches an ent within the document, that attribute is removed, and replaced with the content of the request. If the request URI does not match an attribute in the

document, the server inserts the content of the request as a new attribute in the document, such that the resulting document is compliant to the schema, and such that the request URI, when evaluated, would now point to the attribute which was inserted. There may be more than one way to perform such an insertion; in that case, it is the discretion of the implementor as to how it is done. It may also be possible that the insertion cannot be done without other additional content being inserted, or cannot be done because the new attribute is not compliant to the schema. In such a case, the server MUST return a 409 response code. In all cases, the resulting document MUST be compliant to the schema.

If the creation or insertion was successful, the server returns a 200 OK or 201 Created, as appropriate.

7.3 GET Handling

The semantics of GET are as specified in RFC 2616. This section clarifies the specific content to be returned for a particular URI that represents an XCAP resource.

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 contains a node selector, and that node selector identifies an XML element in an existing document, that element is returned in the 200 response. The content of the response is the portion of the XML document starting with the left bracket of the begin tag of the element, ending with the right bracket of the end tag of the element. If the request URI contains a node selector, and that node selector contains an attribute selector, and that attribute exists in the specified document, the server returns that attribute, formatted as Attribute in the XML 1.0 specifications. In

all cases, if the referenced resource does not exist, a 404 is returned.

7.4 DELETE Handling

The semantics of DELETE are as specified in RFC 2616. This section clarifies the specific content to be deleted for a particular URI that represents an XCAP resource.

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, and the node selector specifies an XML element that exists, that element is

removed from the document. If the document does exist, and the node selector specifies an XML attribute that exists in the document, that attribute is removed from the document. If the node selector returns a no-match, a 404 (Not Found) is returned. However, if removal of the element or attribute would result in 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 Etags

An XCAP server MUST maintain entity tags 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 an entity tag maintained by the server. These entity tags are needed to support conditional PUT and DELETE requests.

When a PUT request is made that creates or replaces a document, the entity tag of that document and all elements and attributes within is updated.

When a PUT request is made to a URI referencing an XML element, the entity tag of that element, its attributes, and all of its enclosed children and their attributes is updated. For a PUT or DELETE request for an XML element, the entity tag of all elements which are

ancestors of that element are updated. However, the entity tags of attributes belonging to elements that are ancestors of the modified element do not have their entity tags changed, because those resources have not actually changed.

When a PUT request is made to a URI referencing an XML attribute, the entity of that attribute is updated. For a PUT or DELETE request for an attribute, the entity tags for its element, and all elements that are ancestors of that element are updated.

8. Examples

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

First, a user Bill creates a new resource-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"?>
<resource-lists xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <list name="friends" uri="sip:friends@example.com" subscribeable="true">
  </list>
</resource-lists>
```

Next, Bill adds an entry to the list:

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

<entry name="Bob" uri="sip:bob@example.com">
  <display-name>Bob Jones</display-name>
</entry>
```

Next, Bill fetches the list:

```
GET
http://xcap.example.com/services/presence-lists/users/bill/fr.xml HTTP/1.1
```

And the result is:


```
HTTP/1.1 200 OK
Etag: "wwhha"
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<resource-lists xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <list name="friends" uri="sip:friends@example.com"
       subscribeable="true">
    <entry name="Bob" uri="sip:bob@example.com">
      <display-name>Bob Jones</display-name>
    </entry>
```

```
  </list>
</resource-lists>
```

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

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

<list name="close-friends" uri="sip:close-friends@example.com"
      subscribeable="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
Etag: "ad88"
Content-Type:text/plain

uri="sip:nancy@example.com"
```

9. Security Considerations

Frequently, the data manipulated by XCAP contains sensitive information. To avoid eavesdroppers from seeing this information, it is RECOMMENDED 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 [10] 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 [11] 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] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[8] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998.

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

[10] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1",

RFC 2817, May 2000.

[11] 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

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

[13] 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.

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

[15] Roach, A., Rosenberg, J. 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.

[16] 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-03 (work in progress), June 2003.

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

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

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

[20] Roach, A., "Session Initiation Protocol (SIP)-Specific Event Notification", RFC 3265, June 2002.

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

obtain a general license or permission for the use of such proprietary rights by implementors or users of this specification can be obtained from the IETF Secretariat.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the IETF Executive Director.

Full Copyright Statement

Acknowledgement

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