The Extensible Markup Language (XML) Configuration Access Protocol (XCAP)
draft-ietf-simple-xcap-03

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 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 [18] authorization policy and presence lists. Presence lists are lists of users whose presence is desired by a watcher [26]. One way to obtain presence information for the list of is to subscribe to a resource which represents that list [21]. In this case, the Resource List Server (RLS) requires access to this list in order to process a SIP [15]SUBSCRIBE [28] request for it. Another way to obtain presence for the users on the list is for a watcher to subscribe to each user individually. In that case, it is convenient to have a server store the list, and when the client boots, it fetches the list from the server. This would allow a user to access their resource lists from different clients.

Requirements for manipulation of presence lists and authorization policies have been specified by the SIMPLE working group [22].

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 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) [25], 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 5). This application usage defines the XML schema [2] 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 [5]. 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 element or attribute within the document. Thus, the HTTP URIs used by XCAP point a document, or to pieces of information that are finer grained than the XML document itself. An HTTP resource which follows the naming conventions and validation constraints defined here is called an XCAP resource.

Since XCAP resources are also HTTP resources, they can be accessed using HTTP methods. Reading an XCAP resource is accomplished with HTTP GET, creating or modifying one is done with HTTP PUT, and removing one of the resources is done with an HTTP DELETE. Properties that HTTP associates with resources, such as entity tags, also apply to XCAP resources. Indeed, entity tags are particularly useful in XCAP, as they allow a number of conditional operations to be performed.

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 [6] and indicate requirement levels for compliant implementations.

4. Definitions

The following terms are used throughout this document:

XCAP Resource: An HTTP resource representing an XML document, an element within an XML document, or an attribute of an element within an XML document that follows the naming and validation constraints of XCAP.

XCAP Server: An HTTP server that understands how to follow the naming and validation constraints defined in this specification.

Application: A collection of software components within a network whose operation depends on data managed and stored on an XCAP server.

Application Usage: Detailed information on the interaction of an application with the XCAP server.

Application Unique ID (AUID): A unique identifier that differentiates XCAP resources accessed by one application from XCAP resources accessed by another.

Naming Conventions: The part of an application usage that specifies well-known URIs used by an application, or more generally, specifies the URIs that are typically accessed by an application during its processing.

XCAP User Identifier (XUI): The XUI is a string, valid as a path element in an HTTP URI, that is associated with each user served by the XCAP server.

XCAP Root: A context that contains all of the documents across all application usages and users that are managed by the server.

Document Selector: A sequence of path segments, with each segment being separated by a "/", that identify the XML document within an XCAP root that is being selected.

Node Selector: A sequence of path segments, with each segment being separated by a "/", that identify the XML node (element or attribute) being selected within a document.

Path Separator: A single path segment equal to two tilde characters (~~) that is used to separate the document selector from the node selector within an HTTP URL.

Document URI: The HTTP URI containing the XCAP root and document selector, resulting in the selection of a specific document. As a result, performing a GET against the document URI would retrieve the document.

Node URI: The HTTP URI containing the XCAP root, document selector, path separator and node selector, resulting in the selection of a specific XML node.

XCAP Root URI: An HTTP URI that representing the XCAP root. Although a valid URI, the XCAP Root URI does not correspond to an actual resource.

Global Tree: A URI that represents the parent for all global documents for a particular application usage within a particular XCAP root.

Home Directory: A URI that represents the parent for all documents for a particular user for a particular application usage within a particular XCAP root.

Positional Insertion: A PUT operation that results in the insertion of a new element into a document such that its position relative to other children of the same parent is set by the client.

5. Application Usages

Each XCAP resource on a server is associated with an application. In order for an application to use those resources, application specific conventions must be specified. Those conventions include the XML schema that defines the structure and constraints of the data, well known URIs to bootstrap access to the data, and so on. All of those application specific conventions are defined by the application usage.

5.1 Application Usage ID (AUID)

Each application usage is associated with a name, called an Application Unique ID (AUID). This name uniquely identifies the application usage, and is different from AUIDs used by other applications. 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 [27] based on the guidelines in Section 13. 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 [11] (and using some of the BNF defined in RFC 2396 [12]) 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
```

5.2 Data Validation

One of the responsibilities of an XCAP server is to validate the content of each XCAP resource when an XCAP client tries to modify one. This is done using two mechanisms. Firstly, all application usages MUST describe their document contents using XML schema [2].

The application usage MUST also identify the MIME type for documents compliant to that schema.

Unfortunately, XML schemas cannot represent every form of data constraint. As an example, one XML element may contain an integer which defines the maximum number of instances of another element. This constraint cannot be represented with XML schema. However, such constraints may be important to the application usage. The application usage defines any additional constraints beyond those in the schema.

Of particular importance are uniqueness constraints. In many cases, an application will require that there only be one instance of some element or attribute within a particular scope. Each uniqueness constraint needs to be specified by identifying the field, or combinations of fields, that need to be unique, and then identifying the scope in which that uniqueness applies. One typical scope is the set of all elements of a certain name within the same parent. Another typical scope is the set of all URIs valid within a particular domain. In some cases these constraints can be specified using XML schema, which provides the <unique> element for this purpose. Other uniqueness constraints, such as URI uniqueness across a domain, cannot be expressed by schema. Whether or not the schema is used to express some of the uniqueness requirements, the application usage MUST specify all uniqueness requirements when it defines its data validation needs.

For example, the resource lists application usage [23] requires that each <list> element have a unique value for the "name" attribute within a single parent. As another example, the RLS services application usage [23] requires that the value of the "uri" attribute of the <service> element be a URI that is unique within the domain of the URI.

Another form of constraint are URI constraints. These are constraints on the scheme or structure of the scheme specific part of the URI. These kinds of constraints cannot be expressed in an XML schema. If these constraints are important to an application usage, they need to be explicitly called out. As an example, the resource lists application usage requires that the URI present in the "uri" attribute of the <entry> element be either a SIP or pres URI [24].

Another important data constraint is referential integrity. Referential integrity is important when the name or value of an element or attribute is used as a key to select another element or attribute. An application usage MAY specify referential integrity constraints. However, XCAP servers are not a replacement for Relational Database Management Systems (RDBMS), and therefore servers are never responsible for maintaining referential integrity. XCAP clients are responsible for making all of the appropriate changes to documents in order to maintain referential integrity.

The data validation information is consumed by both clients, which use them to make sure they construct requests that will be accepted by the server, and by servers, which validate the constraints when they receive a request (with the exception of referential integrity constraints, which are not validated by the server).

5.3 Data Semantics

For each application usage, the data present in the XML document has a well defined semantic. The application usage defines that semantic, so that a client can properly construct a document in order to achieve the desired result. They are not used by the server, as it is purposefully unaware of the semantics of the data it is managing. The data semantics are expressed in English prose by the application usage.

5.4 Naming Conventions

In addition to defining the meaning of the document in the context of a particular application, an application usage has to specify how the applications obtain the documents they need. In particular, it needs to define any well-known URIs used for bootstrapping purposes, and document any other conventions on the URIs used by an application. It should also document how documents reference each other. These conventions are called naming conventions.

As an example, the RLS services application usage allows an RLS to obtain the contents of a resource list when the RLS receives a SUBSCRIBE request for a SIP URI identifying an RLS service. The application usage specifies that the list of service definitions is present within a specific document with a specific name within the global tree. This allows the RLS to perform a single XCAP request to fetch the service definition for the service associated with the SIP URI in a SUBSCRIBE request.

Naming conventions are used by XCAP clients to construct their URIs. The XCAP server does not make use of them.

5.5 Resource Interdependencies

When a user modifies an XCAP resource, the content of many other resources is affected. For example, when a user deletes an XML element within a document, it does so by issuing a DELETE request against the URI for the element resource. However, deleting this element also deletes all child elements and their attributes, each of which is also an XCAP resource. As such, manipulation of one resource affects the state of other resources.

For the most part, these interdependencies are fully specified by the XML schema used by the application usage. However, in some application usages, there is a need for the server to relate resources together, and such a relationship cannot be specified through a schema. This occurs when changes in one document need to affect another document. Typically, this is the case when an application usage is defining a document that acts as a collection of information defined in other documents.

As an example, when a user creates a new RLS service (that is, it creates a new <service> element within an RLS services document), the server adds that element to a read-only global list of services maintained by the server in the global tree. This read-only global list is accessed by the RLS when processing a SIP SUBSCRIBE request.

Resource interdependencies are used by both XCAP clients and servers.

5.6 Authorization Policies

By default, each user is able to access (read, modify, and delete) all of the documents below their home directory, and any user is able to read documents within the global directory. However, only trusted users, explicitly provisioned into the server, can modify global documents.

The application usage can specify a different authorization policy that applies to all documents associated with that application usage. 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.

If an application usage does not wish to change the default authorization policy, it can merely state that the default policy is used.

The authorization policies defined by the application usage are used by the XCAP server during its operation.

5.7 Data Extensibility

An XCAP server MUST understand an application usage in order to process an HTTP request made against a resource for that particular application usage. However, it is not required for the server to understand all of the contents of a document used by an application usage. A server is required to understand the baseline schema defined by the application usage. However, those schemas can define points of extensibility where new content can be added from other namespaces and corresponding schemas. Sometimes, the server will understand those namespaces and therefore have access to their schemas. Sometimes, it will not.

A server MUST allow for documents that contain elements from namespaces not known to the server. In such a case, the server cannot validate that such content is schema compliant; it will only verify that the XML is well-formed.

If a client wants to verify that a server supports a particular namespace before operating on a resource, it can query the server for its capabilities using the XCAP Capabilities application usage, discussed in Section 10.

5.8 Documenting Application Usages

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

- Application Usage ID (AUID): If the application usage is meant for general use on the Internet, the application usage MUST register the AUID into the IETF tree using the IANA procedures defined in Section 13.
- XML Schema
- MIME Type
- Validation Constraints
- Data Semantics
- Naming Conventions
- Resource Interdependencies
- Authorization Policies

5.9 Guidelines for Creating Application Usages

The primary design task when creating a new application usage is to define the schema. Although XCAP can be used with any XML document, intelligent schema design will improve the efficiency and utility of the document when it is manipulated with XCAP.

XCAP provides three fundamental ways to select elements amongst a set of siblings - by the name of the element, by its position, or by the value of a specific attribute. Positional selection always allows a client to get exactly what it wants. However, it requires a client to cache a copy of the document in order to construct the predicate. Furthermore, if a client performs a PUT, it requires the client to reconstruct the PUT processing that a server would follow in order to update its local cached copy. Otherwise, the client will be forced to re-GET the document after every PUT, which is inefficient. As such, it is a good idea to design schemas such that common operations can be performed without requiring the client to cache a copy of the document.

Without positional selection, a client can pick the element at each step by its name or the value of an attribute. Many schemas include elements that can be repeated within a parent (often, minOccurs equals zero or one, and maxOccurs is unbounded). As such, all of the elements have the same name. This leaves the attribute value as the only way to select an element. Because of this, if an application usage expects user to manipulate elements or attributes that are descendants of an element which can repeat, that element SHOULD include, in its schema, an attribute which can be suitably used as a unique index. Furthermore, the naming conventions defined by that application usage SHOULD specify this uniqueness constraint explicitly.

URIs often make a good choice for such unique index. They have fundamental uniqueness properties, and are also usually of semantic significance in the application usage. However, care must be taken when using a URI as an attribute value. URI equality is ususally complex. However, attribute equality is performed by the server using XML rules, which are based on case sensitive string comparison. Thus, XCAP will match URIs based on lexical equality, not functional equality. In such cases, an application usage SHOULD consider these implications carefully.

XCAP provides the ability of a client to operate on a single element, attribute or document at a time. As a result, it may be possible that common operations the client might perform will require a sequence of multiple requests. This is inefficient, and introduces the possibility of failure conditions when another client modifies the document in the middle of a sequence. In such a case, the client will be forced to detect this case using entity tags (discussed below in Section 7.10), and undo its previous changes. This is very difficult.

As a result, the schemas SHOULD be defined so that common operations generally require a single request to perform. Consider an example. Lets say an application usage is defining permissions for users to perform certain operations. The schema can be designed in two ways. The top level of the tree can identify users, and within each user, there can be the permissions associated with the user. In an alternative design, the top level of the tree identifies each permission, and within that permission, the set of users who have it. If, in this application usage, it is common to change the permission for a user from one value to another, the former schema design is better for xcap; it will require a single PUT to make such a change. In the latter case, either the entire document needs to be replaced (which is a single operation), or two PUT operations need to occur - one to remove the user from the old permission, and one to add the user to the new permission.

Naming conventions form another key part of the design of an application usage. The application usage should be certain that XCAP clients know where to "start" to retrieve and modify documents of interest. Generally, this will involve the specification of a well-known document at a well-known URI. That document can contain references to other documents that the client needs to read or modify.

6. URI Construction

In order to manipulate an XCAP resource, the data must be represented by an HTTP URI. XCAP defines a specific naming convention for constructing these URIs. The URI is constructed by concatenating the XCAP root with the document selector with the path separator with a escape coded form of the node selector. The XCAP root is the enclosing context in which all XCAP resources live. The document selector is a path that identifies a document within the XCAP root. The path separator is a path segment with a value of double tilde (~~). It is piece of syntactic sugar that separates the document selector from the node selector. The node selector is an expression that identifies an XML element within a document.

The sections below describe these components in more detail.

6.1 XCAP Root

The root of the XCAP hierarchy is called the XCAP root. It defines the context in which all other resources exist. The XCAP root is represented with an HTTP URI, called the XCAP Root URI. This URI is a valid HTTP URL; however, it doesnt point to any resource that actually exists on the server. Its purpose is to identify 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 root URI within the example.com domain. Typically, the XCAP root URI is provisioned into client devices. A server or domain MAY support multiple XCAP root URIs. In such a case, it is effectively operating as if it were serving separate domains. There is never information carryover or interactions between resources in different XCAP root URIs.

6.2 Document Selector

Each document within the XCAP root is identified by its document selector. The document selector is a sequence of path segments, separated by a slash ("/"). These path segments define a hierarchical structure for organizing documents within any XCAP root. The first path segment MUST be the XCAP AUID. So, continuing the example above, all of the documents used by the resource lists application would be under http://xcap.example.com/services/resource-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, beneath each AUID 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. The subtree beneath "global" is called the global tree. The path segment after the AUID MUST either be "global" or "users".

Within the "users" tree are zero or more sub-trees, each of which identifies documents that apply to a specific user. Each user known to the server is associated with a username, called the XCAP User Identifier (XUI). This XUI MUST be used as the path segment beneath the "users" segment. The subtree beneath an XUI for a particular user is called their home directory. "User" in this context should be interpreted loosely; a user might correspond to device, for example.

XCAP does not itself define what it means for documents to "apply" to a user, beyond specification of a baseline authorization policy, described below in Section 8. Each application usage can specify additional authorization policies which depend on data used by the application itself.

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

The final path segment in the document selector identifies the actual document in the hierarchy. This is equivalent to a filename, except that XCAP does not require that its document resources be stored as files in a file system. However, the term "filename" is used to describe the final path segment in the document selector. In traditional filesystems, the filename would have a filename extension, such as ".xml". There is nothing in this specification that requires or prevents such extensions from being used in the filename. In some cases, the application usage will specify a naming convention for documents, and those naming conventions may or may not specify a file extension. For example, in the RLS services application usage [23], documents in the user's home directory with the filename "index" will be used by the server to compute the global index, which is also a document with the filename "index".

When the naming conventions in an application usage do not constrain the filename conventions (or, more generally, the document selector), an application will know the filename (or more generally, the document selector) because it is included as a reference in a document which is at a well known location. As another example, within the index document defined by RLS services, the <service> element has a child element called <resource-list> whose content is a URI pointing to a resource list within the users home directory.

As a result, if the user creates a new document, and then references that document from a well-known document (such as the index document above), it doesn't matter whether the user includes an extension in the filename or not, as long as the user is consistent and maintains referential integrity. Indeed, it doesn't even matter what the rest of the filename is, independent of whether it has a filename extension or not.

6.3 Node Selector

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

The QName grammar is defined the XML namespaces [3] specification, and the AttValue grammar is defined in the XML specification XML 1.0 [1].

Note that the left bracket, right bracket, and double quote characters, which are meaningful to XCAP, cannot be directly represented in the HTTP URI. As a result, they are escape coded when placed within the HTTP URI.

Similarly, the XML specification defines the QName production for the grammar for element and attribute names, and the AttValue production for the attribute values. Unfortunately, the characters permitted by these productions include some that are not allowed for pchar, which is the production for the allowed set of characters in path segments in the URI. The AttValue production allows many such characters within the US-ASCII set, including the space. Those characters MUST be escaped coded when placed in the URI. Furthermore, QName and

AttValue allow many Unicode characters, outside of US-ASCII. When these characters need to be represented in the HTTP URI, they are escape coded. To do this, the data should be encoded first as octets according to the UTF-8 character encoding [17] and then only those octets that do not correspond to characters in the unreserved set should be percent-encoded. For example, the character A would be represented as "A", the character LATIN CAPITAL LETTER A WITH GRAVE would be represented as "%C3%80", and the character KATAKANA LETTER A would be represented as "%E3%82%A2".

As a result, the grammar above represents the expressions processed by the XCAP server internally after it has un-escape-coded the URL. The on-the-wire format is dictated by RFC 2396 [12]. In the discussions an examples below, the node selectors are presented in their internal format prior to encoding when not part of an HTTP URL. If an example includes a node selector within an HTTP URL, it is presented in its escape coded form.

The node selector is based on the concepts in XPath [9]. Indeed, the node selector expression, before it is escape coded for representation in the HTTP URL, 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 much unneeded complexity into XCAP.

To determine 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 single 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, by name and position, or by name, position and attribute. In all cases, the name can be wildcarded, so that all elements get selected.

The selection operation operates as follows. Within the current document context, the children of that context are enumerated in document order. If the context is the document, its child is the root element in the document. If the context is an element, its children are all of the children of that element (naturally). Next, those elements whose name is not a match for NameorAny are discarded. An elements name is a match if NameorAny is the wildcard, or, if its not a wildcard, the element name matches NameorAny. Matching is discussed below. The result is an ordered list of elements.

The elements in the list are further filtered by the predicates, which are the expressions in square brackets following NameorAny. Each predicate further prunes the elements from the current ordered list. These predicates are evaluated in order. If the content of the content is a position, the position-th element is selected, and all others are discarded. If there are fewer elements in the list than the value of position, the result is a no-match.

If the content of the predicate is an attribute name and value, all elements possessing that attribute with that value are selected, and all others are discarded. Note that elements cannot be selected based on any namespace attributes. That is, a step like el-name[@xmlns="namespace"] will never match an element, even if there is an element in the list that specifies a default namespace of "namespace". If there are no elements with attributes having the given name and value, the result is a no-match.

After the predicates have been applied, the result will be a no-match, one element, or multiple elements. If the result is multiple elements, the node selector is invalid. Each step in a node selector MUST produce a single element to form the context for the next step. This is more restrictive than general XPath expressions, which allow a context to contain multiple elements. If the result is a no-match, the node selector is invalid. The node selector is only valid if a single element was selected. This element becomes the context for the evaluation of the next step in the node selector expression.

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 in the element in the current context. If there is not, the result is considered a no-match. Otherwise, that attribute is selected. Note that namespace attributes (such as xmlns) cannot be selected.

As a result, once the entire node selector is evaluated against the document, the result will either be a no-match, invalid, or a single element or single attribute.

Matching of element names and attributes is performed as follows. All elements and attributes are expanded as described in XML namespaces [3]. The element and attribute names in the step being evaluated are also expanded. This expansion form requires that any namespace prefixes in the QName be expanded. This expansion is done using the namespace bindings that are in scope for the current context element. Like attributes, if the QName within a step has no namespace prefix, the namespace URI is null. The comparisons are then done against these expanded forms [[TODO: This is the rules in XPath, but I dont see how they can work. Need to check up on that.]]

Comments, text content, and processing declarations in the XML document cannot be selected by the expressions defined here. Of course, if such information is present in a document, and a user selects an XML element enclosing that data, that information would be included in a resulting GET, for example.

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>
```

Example XML Document

Figure 3

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>
```

7. 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.

In all cases where the client modifies a document, by deleting or inserting a document, element or attribute resource, the client SHOULD verify that, if the operation were to succeed, the resulting document would meet the data constraints defined by the application usage, including schema validation. For example, if the client performs a PUT operation to http://xcap.example.com/rls-services/users/joe/mybuddies, rls-services is the application unique ID, and the constraints defined by it SHOULD be followed.

The client will know what URI to use based on the naming conventions described by the application usage.

If the document, after modification, does not meet the data constraints, the server will reject it with a 409. The 409 response may contain an XML body, formatted according to the schema in Section 9.2, which provides further information on the nature of the error. The client MAY use this information to try and alter the request so that this time, it might succeed. The client SHOULD NOT simply retry the request without changing some aspect of it.

In some cases, the application usage will dictate involve a uniqueness constraint that the client cannot guarantee on its own. One such example is that a URI has to be unique within a domain. Typically, the client is not the owner of the domain, and so it cannot be sure that a URI is unique. In such a case, the client can either generate a sufficiently random identifier, or it can pick a "vanity" identifier in the hopes that it is not taken. In either case, if the identifier is not unique, the server will reject the request with a 409 and suggest alternatives that the client can use to try again. If the server does not suggest alternatives, the client SHOULD attempt to use random identifiers with increasing amounts of randomness.

HTTP also specifies that PUT and DELETE requests are idempotent. This means that, if the client performs a PUT on a document and it succeeds, it can perform the same PUT, and the resulting document will look the same. Similarly, when a client performs a DELETE, if it succeeds, a subsequent DELETE to the same URI should find the resource deleted and thus have no further effect. To maintain this property, the client SHOULD construct its URIs such that, after the modification has taken place, the URI in the request will point to the resource just inserted for PUT (i.e., the body of the request), and will point to nothing for DELETE. If this property is maintained, it is the case that GET to the URI in the PUT will return the same content (i.e., GET(PUT(X)) == x). This property is synonymous with idempotency. If the client's request does not have this property, the server will reject the request with a 409 and indicate a cannot-insert error condition.

If the result of the PUT is a 200 or 201 response, the operation was successful. Other response codes to any request, such as a redirection, are processed as per RFC 2616 [5].

7.1 Create or Replace a Document

To create or replace a document, the client constructs a URI that references the location where the document is to be placed. This URI MUST be a document URI, and therefore contain the XCAP root and document selector. The client then invokes a PUT method on that URI.

The MIME content type MUST be the type defined by the application usage. For example, it would be "application/rls-services+xml" for an RLS services [23] document, and not "application/xml".

If the Request-URI identifies a document that already exists in the server, the PUT operation replaces that document with the content of the request. If the Request-URI does not identify an existing document, the document is created on the server at that specific URI.

7.2 Delete a Document

To delete a document, the client constructs a URI that references the document to be deleted. This URI MUST be a document URI. The client then invokes a DELETE operation on the URI to delete the document.

7.3 Fetch 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 URI.

7.4 Create or Replace an Element

To create or replace an XML element within an existing document, the client constructs a URI whose document selector points to the document to be modified. The node selector MUST be present in the URI, separated from the document selector with the path separator. To create this this element within the document, the node selector is constructed such that it is a no-match against the current document, but if the element in the body of the request was added to the document as desired by the client, the node selector would select that element. To replace an element in the document, the node selector is constructed so that it is a match against the element in the current document to be replaced, as well as a match to the new element (present in the body of the PUT request) that is to replace it.

Oftentimes, the client will wish to insert an element into a document in a certain position relative to other children of the same parent. This is called a positional insertion. They often arise because the schema constrains where the element can occur, or because ordering of elements is significant within the schema. To accomplish this, the client can use a node selector of the following form:

```
parent/*[position][unique-attribute-value]
```

Here, "parent" is an expression for the parent of the element to be inserted. "position" is the position amongst the existing children of this parent where the new element is to be inserted. "unique-attribute-value" is an attribute name and value for the element to be inserted, which is different from the current element in "position". The second predicate is needed so that the overall expression is a no-match when evaluated against the current children. Otherwise, the PUT would replace the existing element in that position.

Consider the example document in Figure 3. The client would like to insert a new <watcher> element as the second element underneath <watcher-list>. However, it cannot just PUT to a URI with the watcherinfo/watcher-list/*[2] node selector; this node selector would select the existing 2nd child of <watcher-list> and replace it. Thus, the PUT has to be made to a URI with watcherinfo/watcher-list/*[2][@id="hhggff"] as the node selector, where "hhggff" is the value of the "id" attribute of the new element to be inserted. This node-selector is a no-match against the current document, and would be a match against the new element if it was inserted as the 2nd child of <watcher-list>.

The "*" indicates that all children of <watcher-info> are to be considered when computing the position for insertion. If, instead of a *, an element name was present, the expression above would insert the new element as the position-th element amongst those with the same name.

Once the client constructs the URI, it invokes the HTTP PUT method.

The content in the request MUST be an XML element. Specifically, it contains the element, starting with the opening bracket for the begin tag for that element, including the attributes and content of that element (whether it be text or other child elements), and ending with the closing bracket for the end tag for that element. The MIME type in the request MUST be "application/xcap-el+xml", defined in Section 13.2.1. If the node selector, when evaluated against the current document, results in a no-match, the server performs a creation operation. If the node selector, when evaluated against the current document, is a match for an element in the current document, the server replaces it with the content of the PUT request. This replacement is complete; that is, the old element (including its attributes and content) are removed, and the new one, including its attributes and content, is put in its place.

To be certain that element insertions are idempotent, the client can check that the attribute predicates in the final path segment of the URI match the attributes of the element in the body of the request. As an example of an request that would not be idempotent, consider the following PUT request:

```
   PUT
   http://xcap.example.com/rls-services/users/bill/index/~~/rls-
services/service%5b@uri=%22sip:good-friends@example.com%5d
    HTTP/1.1
   Content-Type:application/xcap-el+xml

   <service uri="sip:mybuddies@example.com">
     <resource-list>http://xcap.example.com/resource-
lists/users/joe/index/~~/resource-lists/list%5b@name=%22l1%22%5d
   </resource-list>
     <packages>
      <package>presence</package>
     </packages>
   </service>
```

This request will fail with a 409. The Request URI contains a final path segment with a predicate based on attributes - @uri="sip:good-friends@example.com". However, this will not match the value of the "uri" attribute in the element in the body.

When the client does not explicitly indicate a position in which to insert a new element, the server will insert that element as the last child of that parent. If this is not the desired position, the client should perform a positional insertion.

7.5 Delete an Element

To delete an element from a document, the client constructs a URI whose document selector points to the document containing the element to be deleted. The node selector MUST be present following the path separator, 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 (including its attributes and its content, such as any children).

7.6 Fetch an Element

To fetch an element of a document, the client constructs a URI whose document selector points to the document containing the element to be fetched. The node selector MUST be present following the path separator, and must identify the element to be fetched.

The client then invokes the GET method. The 200 OK 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. This will, as a result, include all attributes and child elements of that element.

7.7 Create or Replace an Attribute

To create or replace an attribute in an existing element of a document, the client constructs a URI whose document selector points to the document to be modified. The node selector, following the path separator, MUST be present. The node selector MUST be constructed such that, if the attribute was created or replaced as desired, the node selector would select that attribute. If the node selector, when evaluated against the current document, results in a no-match, it is a creation operation. If it matches an existing attribute, it is a replacement operation.

The client then invokes the HTTP PUT method. The content defined by the request MUST be the value of the attribute, compliant to the grammar for AttValue as defined in XML 1.0. Note that, unlike when AttValue is present in the URI, there is no escape coding. Escaping only applies to URIs. This request MUST be sent with the Content-Type of "application/xcap-att+xml" as defined in Section 13.2.2. 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.

To be certain that attribute insertions are idempotent, the client can check that any attribute predicate in the path segment that selects the element into which the attribute is inserted, matches a different attribute than the one being inserted by the request. As an example of a request that would not be idempotent, consider the following PUT request:

```
   PUT
   http://xcap.example.com/rls-services/users/bill/index/~~/rls-
services/service%5b@uri=%22sip:good-friends@example.com%5d/@uri
    HTTP/1.1
   Content-Type:application/xcap-att+xml

   "sip:bad-friends@example.com"
```

This request will fail with a 409.

7.8 Delete an Attribute

To delete attributes from the document, the client constructs a URI whose document selector points to the document containing the attributes to be deleted. The node selector MUST be present following the path separator, 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.

7.9 Fetch an Attribute

To fetch an attribute of a document, the client constructs a URI whose document selector points to the document containing the attribute to be fetched. The node selector MUST be present following the path separator, containing an expression identifying the attribute whose value is to be fetched.

The client then invokes the GET method. The 200 OK response will contain an "application/xcap-att+xml" document with the specified attribute, formatted according to the grammar of AttValue as defined in the XML 1.0 specifications.

7.10 Conditional Operations

The HTTP specification defines several header fields that can be used by a client to make the processing of the request conditional. In particular, the If-None-Match and If-Match header fields allow a client to make them conditional on the current value of the entity tag for the resource. These conditional operations are particularly useful for XCAP resources.

For example, it is anticipated that clients will frequently wish to cache the current version of a document. So, when the client starts up, it will fetch the current document from the server and store it. When it does so, the GET response will contain the entity tag for the document resource. Each resource within a document maintained by the server will share the same value of the entity tag. As a result, the entity tag returned by the server for the document resource is applicable to element and attribute resources within the document.

If the client wishes to modify an element or attribute within the document, but it wants to be certain that the document hasn't been modified since the client last operated on it, it can include an If-Match header field in the request, containing the value of the entity tag known to the client for all resources within the document. If the document has changed, the server will reject this request with a 412 response. In that case, the client will need to flush its cached version, fetch the entire document, and store the new entity tag returned by the server in the 200 OK to the GET request.

Its important to note that entity tags are defined for each resource in the document, even though they share the same value. As a result, if a resource does not exist in the document, there is no entity tag for it, and a PUT to such a resource with an If-Match header field will always fail with a 412. For this reason, PUT operations that insert a new element or attribute into a document cannot be conditioned on the entity tag for the document itself. If the client wishes to make sure that the document has not changed before adding the new element or attribute, the client can "pop up a level", and instead of inserting a new element or attribute, it can modify the parent of that new element and attribute such that the new value contains the new element or attribute.

In another example, a client may wish to insert a new element into a document, but wants to be sure that the insertion will only take place if that element does not exist. In other words, the client wants the PUT operation to be a creation, not a replacement. To accomplish that, the client can insert the If-None-Match header field into the PUT request, with a value of *. This tells the server to reject the request with a 412 if resource exists.

As another example, a when a client fetches a document, and there is an older version cached, it is useful for clients to use a conditional GET in order to reduce network usage if the cached copy is still valid. This is done by including, in the GET request, the If-None-Match header field with a value equal to the current etag held by the client for the document. The server will only generate a 200 OK reponse if the etag held by the server differs than that held by the client. If it doesnt differ, the server will respond with a 304 response.

8. 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.

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 implement HTTP Digest [10]. Furthermore, servers MUST implement HTTP over TLS, RFC 2818 [13]. It is RECOMMENDED that administrators use an HTTPS URI as the XCAP root 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 application usage defines the authorization policies. An application usage may specify that the default is used. This default is described in Section 5.6.

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

8.1 POST Handling

XCAP resources do not represent processing scripts. As a result, POST operations to HTTP URIs representing XCAP resources are not defined. A server receiving such a request for an XCAP resource SHOULD return a 405.

8.2 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.

8.2.1 Locating the Parent

The first step the server performs is to locate the parent, whether it is a directory or element, in which the resource is to be placed. To do that, the server removes the last path segment from the URI. The rest of the URI refers to the parent. This parent can be a document, element, or prefix of a document selector (called a directory, even though this specification does not mandate that doocuments are actually stored in a filesystem). This URI is called the parent URI. The path segment that was removed is called the target selector, and the node (element, document or attribute) it describes is called the target node.

If the parent URI has no path separator, it is referring to the directory into which the document should be inserted. If this directory does not exist, the server MUST return a 409 response, and SHOULD include a detailed conflict report including the <no-parent> element. Detailed conflict reports are discussed in Section 9. If the directory does exist, the server checks to see if there is a document with the same filename as the target node. If there is, the operation is the replacement operation discussed in Section 8.2.4. If it does not exist, it is the creation operation, discussed in Section 8.2.4.

If the parent URI has a path separator, the document selector is extracted, and that document is retrieved. If the document does not exist, the server MUST return a 409 response, and SHOULD include a detailed conflict report including the <no-parent> element. If it does exist, the node selector is extracted, and unescaped (recall that the node selector is escape coded). The node selector is applied to the document based on the matching operations discussed in Section 6.3. If the result is a no-match or invalid, the server MUST return a 409 response, and SHOULD include a detailed conflict report including the <no-parent> element.

If the node-selector is valid, the server examines the target selector, and evaluates in within the context of the parent node. If the target node exists within the parent, the operation is a replacement, as described in Section 8.2.4. If it does not exist, it is the creation operation, discussed in Section 8.2.4.

Before performing the replacement or creation, as determined based on the logic above, the server validates the content of the request as described in Section 8.2.2

8.2.2 Verifying Document Content

If the PUT request is for a document (the request URI had no path separator), the content of the request body has to be a well-formed XML document. If it is not, the server MUST reject the request with a 409 response code. That response SHOULD include a detailed conflict report including the <not-well-formed> element. If the MIME type in the Content-Type header field of the request is not equal to the MIME type defined for the application usage, the server MUST reject the request with a 415.

If the PUT request is for an element, the content of the request body has to be a well-balanced region of an XML document, also known as an XML fragment body in The XML Fragment Interchange [4] specification, including only a single element. If it is not, the server MUST reject the request with a 409 response code. That response SHOULD include a detailed conflict report including the <not-xml-frag> element. If the MIME type in the Content-Type header field of the request is not equal to "application/xcap-el+xml", the server MUST reject the request with a 415.

If the PUT request is for an attribute, the content of the request body has to be a sequence of characters that comply with the grammar for AttValue as defined above. If it is not, the server MUST reject the request with a 409 response code. That response SHOULD include a detailed conflict report including the <not-xml-att-value> element. If the MIME type in the Content-Type header field of the request is not equal to "application/xcap-att+xml", the server MUST reject the request with a 415.

8.2.3 Creation

The steps in this sub-section are followed if the PUT request will result in the creation of a new document, element or attribute.

If the PUT request is for a document, the content of the request body is placed into the directory, and its filename is associated with the target node, which is a document.

If the PUT request is for an element, the server inserts the content of the request body as a new child element of the parent element selected in Section 8.2.1. The insertion is done such that, the request URI, when evaluated, would now point to the element which was inserted. If the target selector is defined by a by-name or by-attr production (in other words, there is no position indicated) the server MUST insert the element after any other siblings. If a position is indicated, the server MUST insert the element so that it is the position-th element amongst all siblings whose name matches NameorAny.

It is possible that the element cannot be inserted such that the request URI, when evaluated, returns the content provided in the request. Such a request is not idempotent, and is not allowed for PUT. This happens when the element in the body is not described by the expression in the target selector. An example of this case is described in Section 7.4. If this happens the server MUST NOT perform the insertion, and MUST reject the request with a 409 response. The body of the response SHOULD contain a detailed conflict report containing the <cannot-insert> element. It is important to note that schema compliance does not play a role while performing the insertion. That is, the decision of where the element gets inserted is dictated entirely by the structure of the request-URI, the current document, and the rules in this specification.

If the PUT request is for an attribute, the server inserts the content of the request body as the value of the attribute. The name of the attribute is equal to the att-name from the attribute-selector in the target selector.

Assuming that the insertion can be accomplished, the server verifies that the insertion results in a document that meets the constraints of the application usage. This is dicussed in Section 8.2.5.

8.2.4 Replacement

The steps in this sub-section are followed if the PUT request will result in the replacement of a document, element or attribute with the contents of the request.

If the PUT request is for a document, the content of the request body is placed into the directory, replacing the document with the same filename.

If the PUT request is for an element, the server replaces the target node with the content of the request body. As in the creation case, it is possible that, after replacement, the request URI does not select the element that was just inserted. If this happens the server MUST NOT perform the replacement, and MUST reject the request with a 409 response. The body of the response SHOULD contain a detailed conflict report containing the <cannot-insert> element.

If the PUT request is for an attribute, the server sets the value of the selected attribute to the content of the request body. It is possible in the replacement case (but not in the creation case), that, after replacement of the attribute, the request URI no longer selects the attribute that was just replaced. The scenario in which this can happen is discussed in Section 7.7. If this is the case, the server MUST NOT perform the replacement, and MUST reject the request with a 409 response. The body of the response SHOULD contain a detailed conflict report containing the <cannot-insert> element.

8.2.5 Validation

Once the document, element or attribute has been tentatively inserted, the server needs to verify that the resulting document meets the data constraints outlined by the application usage.

First, the server checks that the final document is compliant to the schema. If it is not, the server MUST NOT perform the insertion. It MUST reject the request with a 409 response. That response SHOULD contain a detailed conflict report containing the <schema-validation-error> element.

Next, the server checks for any uniqueness constraints identified by the application usage. If the application usage required that a particular element or attribute had a unique value within a specific scope, the server would check that this uniqueness property still exists. If the application usage required that a URI within the document was unique within the domain, the server checks whether it is the case. If any of these uniqueness constraints are not met, the server MUST NOT perform the insertion. It MUST reject the request with a 409 response. That response SHOULD contain a detailed conflict report containing the <uniqueness-failure> element. That element can contain suggested values that the client can retry with. These SHOULD be values that, at the time the server generates the 409, would meet the uniqueness constraints.

The server also checks for URI constraints and other non-schema data constraints. If the document fails one of these constraints, the server MUST NOT perform the insertion. It MUST reject the request with a 409 response. That response SHOULD contain a detailed conflict report containing the <constraint-failure> element. That element indicates that the document failed non-schema data constraints explicitly called out by the application usage.

8.2.6 Resource Interdependencies

Because XCAP resources include elements, attributes and documents, each of which has its own HTTP URI, the creation or modification of one resource affects the state of many others. For example, insertion of a document creates resources on the server for all of the elements and attributes within that document. After the server has performed the insertion associated with the PUT, the server MUST create and/or modify those resources affected by that PUT. The structure of the document completely defines the inter-relationship between those resources. Normally a server will not need to actually do anything to meet this requirement, since those other resources would normally be resolved dynamically when requests are made against them.

However, the application usage can specify other resource inter-dependencies. The server MUST create or modify the resources specified by the application usage.

If the creation or insertion was successful, and the resource interdependcies are resolved, the server returns a 200 OK or 201 Created, as appropriate. This response MUST not contain any content.

8.3 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 selector, the server returns the document specified by the URI if it exists, else returns a 404 response. The MIME type of the body of the 200 OK response MUST be the MIME type defined by that application usage (i.e., "application/resource-lists+xml").

If the request URI contains a node selector, the server obtains the document specified by the document selector, and if it is found, evaluates the node-selector within that document. If no document is found, or if the node-selector is a no-match or invalid, the server returns a 404 response. Otherwise, the server returns a 200 OK response. If the node selector identifies an XML element, that element is returned in the 200 OK response as an XML fragment body containing the selected element. The MIME type of the response MUST be "application/xcap-el+xml". If the node selector identifies an XML attribute, the value of that attribute is returned in the body of the response. The MIME type of the response MUST be "application/xcap-att+xml".

8.4 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 selector, the server deletes the document specified by the URI if it exists and returns a 200 OK, else returns a 404 response.

If the request URI contains a node selector, the server obtains the document specified by the document selector, and if it is found, evaluates the node-selector within that document. If no document is found, or if the node-selector is a no-match or invalid, the server returns a 404 response. Otherwise, the server removes the specified element or attribute from the document and performs the validation checks defined in Section 8.2.2. If the deletion will cause a failure of one of the constraints, the deletion MUST NOT take place. The server follows the procedures in Section 8.2.2 for computing the 409 response. If the deletion results in a document that is still valid, the server MUST perform the deletion and return a 200 OK response.

Before the server returns the 200 OK response to a DELETE, it MUST process the resource interdependcies as defined in Section 8.2.6.

8.5 Managing Etags

An XCAP server MUST maintain entity tags for all resources that it maintains. This specification introduces the additional constraint that when one resource within a document (including the document itself) changes, that resource is assigned a new etag, and all other resources within that document MUST be assigned the same etag value. An XCAP server MUST include the Etag header field in all 200 or 201 responses to PUT, GET, or DELETE [[Todo: Not sure we'll see them in DELETE responses. Need to check.]]. XCAP resources do not introduce new requirements on the strength of the entity tags; as in RFC 2616, weak ones MAY be used if performance constraints or other conditions make usage of strong ones untenable for some reason.

As a result of this constraint, when a client makes a change to an element or attribute within a document, the response to that operation will convey the entity tag of the resource that was just affected. Since the client knows that this entity tag value is shared by all of the other resources in the document, the client can make conditional requests against other resources using that entity tag.

9. Detailed Conflict Reports

In cases where the server returns a 409 error response, that response will usually include a document in the body of the response which provides further details on the nature of the error. This document is an XML document, formatted according to the schema of Section 9.2. Its MIME type, registered by this specification, is "application/xcap-error+xml".

9.1 Document Structure

The document structure is simple. It contains the root element <xcap-error>. The content of this element is a specific error condition. Each error condition is represented by a different element. This allows for different error conditions to provide different data about the nature of the error. All error elements support a "phrase" attribute, which can contain text meant for rendering to a human user.

The following error elements are defined by this specification:

<not-well-formed>: This indicates that the body of the request was not a well-formed XML document.

<not-xml-frag>: This indicates that the request was supposed to contain a valid XML fragment body, but did not. Most likely this is because the XML in the body was malformed or not balanced.

<no-parent>: This indicates that an attempt to insert an element, attribute or document failed because the document or element into which the insertion was supposed to occur does not exist. This error element can contain an optional <ancestor> element, which provides an HTTP URI of the xcap resource that identifies the closest ancestor element that does exist in the document. Because this is a valid HTTP URI, its node selector component MUST be escape encoded.

<schema-validation-error>: This element indicates that the document was not compliant to the schema after the requested operation was performed.

<not-xml-att-value>: This indicates that the request was supposed to contain a valid XML attribute value, but did not.

<cannot-insert>: This indicates that the requested PUT operation could not be performed because it would not be idempotent.

<uniqueness-failure>: This indicates that the requested operation would result in a document that did not meet a uniqueness constraint defined by the application usage. For each URI, element or attribute specified by the client which is not unique, an <exists> element is present as the content of the error element. Each <exists> element has a "field" attribute that contains the node selector identifying the XML element or attribute whose value needs to be unique, but wasn't. Note that

the double quote character, which is allowed in node selectors, cannot appear within the value of an attribute. As such, it MUST be represented as &quot. Beyond that, since the node selector is not appearing within an HTTP URL, there is no escape encoding. The <exists> element can optionally contain a list of suggested alternate values which do not currently exist on the server.

<constraint-failure>: This indicates that the requested operation would result in a document that failed a data constraint defined by the application usage, but not enforced by the schema or a uniqueness constraint.

Extensions to XCAP can define additional error elements.

As an example, the following document indicates that the user attempted to create an RLS service using the URI sip:friends@example.com, but that URI already exists:

```
<?xml version="1.0" encoding="UTF-8"?>
<xcap-error xmlns="urn:ietf:params:xml:ns:xcap-error">
 <uniqueness-failure>
  <exists field="rls-services/service/@uri">
    <alt-value>sip:mybuddies@example.com</alt-value>
  </exists>
 </uniqueness-failure>
</xcap-error>
```

9.2 XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="urn:ietf:params:xml:ns:xcap-error"
  xmlns="urn:ietf:params:xml:ns:xcap-error"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
 <xs:element name="error-element" abstract="true"/>
 <xs:element name="xcap-error">
  <xs:annotation>
   <xs:documentation>Indicates the reason for the error.</xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="error-element"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
```

```
    <xs:element name="schema-validation-error" substitutionGroup="error-
element">
     <xs:annotation>
      <xs:documentation>This element indicates
   that the document was not compliant to the schema after the requested
   operation was performed.</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:attribute name="phrase" type="xs:string" use="optional"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="not-xml-frag" substitutionGroup="error-element">
     <xs:annotation>
      <xs:documentation>This indicates that the request was supposed to
   contain a valid XML fragment body, but did not.</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:attribute name="phrase" type="xs:string" use="optional"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="no-parent" substitutionGroup="error-element">
     <xs:annotation>
      <xs:documentation>This indicates that an attempt to insert
   an element, attribute or document failed because the document or
   element into which the insertion was
   supposed to occur does not exist</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ancestor" type="xs:anyURI" minOccurs="0">
        <xs:annotation>
         <xs:documentation>Contains an HTTP URI that points to the
   element which is the closest ancestor that does exist.</xs:documentation>
        </xs:annotation>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="phrase" type="xs:string" use="optional"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="cannot-insert" substitutionGroup="error-element">
     <xs:annotation>
      <xs:documentation>This indicates that the requested
   PUT operation could not be performed because it would not be
   idempotent.</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:attribute name="phrase" type="xs:string" use="optional"/>
     </xs:complexType>
    </xs:element>
```

```
    <xs:element name="not-xml-att-value" substitutionGroup="error-element">
     <xs:annotation>
      <xs:documentation>This indicates that the
   request was supposed to contain a valid XML attribute value, but did
   not.</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:attribute name="phrase" type="xs:string" use="optional"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="uniqueness-failure" substitutionGroup="error-element">
     <xs:annotation>
      <xs:documentation>This indicates that the
   requested operation would result in a document that did not meet a
   uniqueness constraint defined by the application usage.</xs:documentation>
     </xs:annotation>
     <xs:complexType>
      <xs:sequence>
       <xs:element name="exists" maxOccurs="unbounded">
        <xs:annotation>
         <xs:documentation>For each URI,
   element or attribute specified by the client which is not unique,
   one of these is present.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
         <xs:sequence minOccurs="0">
          <xs:element name="alt-value" type="xs:string" maxOccurs="unbounded">
           <xs:annotation>
            <xs:documentation>An optional set of alternate values
   can be provided.</xs:documentation>
           </xs:annotation>
          </xs:element>
         </xs:sequence>
         <xs:attribute name="field" type="xs:string" use="required"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="phrase" type="xs:string" use="optional"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="not-well-formed" substitutionGroup="error-element">
     <xs:annotation>
      <xs:documentation>This indicates that the body of the request was
   not a well-formed document.</xs:documentation>
     </xs:annotation>
    </xs:element>
    <xs:element name="constraint-failure" substitutionGroup="error-element">
     <xs:annotation>
```

```
      <xs:documentation>This indicates that the
   requested operation would result in a document that failed a data
   constraint defined by the application usage, but not enforced by the
   schema or a uniqueness constraint.</xs:documentation>
     </xs:annotation>
    </xs:element>
   </xs:schema>
```

10. XCAP Server Capabilities

XCAP can be extended through the addition of new application usages and extensions to the core protocol. An XCAP server can also be extended to support new namespaces. It will often be necessary for a client to determine what extensions, application usages or namespaces a server supports before making a request. To enable that, this specification defines an application usage with the AUID "xcap-caps". All XCAP servers MUST support this application usage. This usage defines a single document within the global tree which lists the capabilities of the server. Clients can read this well-known document, and therefore learn the capabilities of the server.

The structure of the document is simple. The root element is <xcap-caps>. Its children are <auids>, <extensions>, and <namespaces>. Each of these contain a list of AUIDs, extensions and namespaces supported by the server. Extensions are named by tokens defined by the extension. Namespaces are identified by their namespace URI. Since all XCAP servers support the "xcap-caps" AUID, it MUST be listed in the <auids> element.

The following sections provide the information needed to define this application usage.

10.1 Application Usage ID (AUID)

This specification defines the "xcap-caps" AUID within the IETF tree, via the IANA registration in Section 13.

10.2 XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="urn:ietf:xml:params:ns:xcap-caps"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="urn:ietf:xml:params:ns:xcap-caps"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
 <xs:element name="xcap-caps">
  <xs:annotation>
   <xs:documentation>Root element for xcap-caps</xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:sequence>
    <xs:element name="auids">
     <xs:annotation>
      <xs:documentation>List of supported AUID.</xs:documentation>
     </xs:annotation>
```

```
        <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="auid" type="auidType"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="extensions">
        <xs:annotation>
         <xs:documentation>List of supported extensions.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="extension" type="extensionType"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="namespaces">
        <xs:annotation>
         <xs:documentation>List of supported namespaces.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="namespace" type="namespaceType"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:simpleType name="auidType">
     <xs:annotation>
      <xs:documentation>AUID Type</xs:documentation>
     </xs:annotation>
     <xs:restriction base="xs:string"/>
    </xs:simpleType>
    <xs:simpleType name="extensionType">
     <xs:annotation>
      <xs:documentation>Extension Type</xs:documentation>
     </xs:annotation>
     <xs:restriction base="xs:string"/>
    </xs:simpleType>
    <xs:simpleType name="namespaceType">
     <xs:annotation>
      <xs:documentation>Namespace type</xs:documentation>
     </xs:annotation>
     <xs:restriction base="xs:anyURI"/>
    </xs:simpleType>
```

```
</xs:schema>
```

10.3 MIME Type

Documents conformant to this schema are known by the MIME type "application/xcap-caps+xml", registered in Section 13.2.4.

10.4 Validation Constraints

There are no additional validation constraints associated with this application usage.

10.5 Data Semantics

Data semantics are defined above.

10.6 Naming Conventions

A server MUST maintain a single instance of the document in the global tree, using the filename "index". There MUST NOT be an instance of this document in the users tree.

10.7 Resource Interdependencies

There are no resource interdependencies in this application usage beyond those defined by the schema.

10.8 Authorization Policies

This application usage does not change the default authorization policy defined by XCAP.

11. Examples

This section goes through several examples, making use of the resource-lists and rls-services [23] XCAP application usages.

First, a user Bill creates a new document (see Section 7.1). This document is a new resource-list, initially with a single list, called friends, with no users in it:

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

<?xml version="1.0" encoding="UTF-8"?>
<resource-lists xmlns="urn:ietf:params:xml:ns:resource-lists">
  <list name="friends">
  </list>
</resource-lists>
```

Next, Bill creates an RLS services document defining a single RLS service referencing this list. This service has a URI of sip:myfriends@example.com:

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

<?xml version="1.0" encoding="UTF-8"?>
<rls-services xmlns="urn:ietf:params:xml:ns:rls-services"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<service uri="sip:myfriends@example.com">
  <resource-list>http://xcap.example.com/services/resource-
lists/users/bill/fr.xml/~~/resource-lists/list%5b@name=%22friends%22%5d
</resource-list>
  <packages>
   <package>presence</package>
  </packages>
 </service>
</rls-services>
```

Next, Bill creates an element in the resource-lists document (Section 7.4). In particular, he adds an entry to the list:

```
PUT
http://xcap.example.com/services/resource-lists/users/bill/fr.xml
/~~/resource-lists/list%5b@name=%22friends%22%5d/entry HTTP/1.1
Content-Type:application/xcap-el+xml

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

Next, Bill fetches the document (Section 7.3):

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

And the result is:

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

<?xml version="1.0" encoding="UTF-8"?>
<resource-lists xmlns="urn:ietf:params:xml:ns:resource-lists"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <list name="friends">
     <entry 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. This is another element creation (Section 7.4):

```
PUT
http://xcap.example.com/services/resource-lists/users/bill/fr.xml/~~/
resource-lists/list%5b@name=%22friends%22%5d/list%5b@name=%22close-
friends%22%5d HTTP/1.1
Content-Type: application/xml-fragment-body

<list name="close-friends">
   <entry uri="sip:joe@example.com">
     <display-name>Joe Smith</display-name>
   </entry>
   <entry uri="sip:nancy@example.com">
     <display-name>Nancy Gross</display-name>
   </entry>
   <entry 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 (Section 7.5):

```
DELETE
http://xcap.example.com/services/resource-lists/users/bill/fr.xml/
~~/resource-lists/list/list/entry%5b@uri=%22sip:petri@example.com%22%5d
HTTP/1.1
```

Bill decides to check on the URI for Nancy, so he fetches a particular attribute (Section 7.6):

```
GET
http://xcap.example.com/services/resource-lists/users/bill/fr.xml/
~~/resource-lists/list/list/entry%5b2%5d/@uri HTTP/1.1
```

and the server responds:

```
HTTP/1.1 200 OK
Etag: "ad88"
Content-Type:application/xcap-att+xml

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

12. Security Considerations

Frequently, the data manipulated by XCAP contains sensitive information. To avoid eavesdroppers from seeing this information, it is RECOMMENDED that an administrator hand out an https URI as the XCAP root 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 [14] 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 [10] to establish its identity, and this SHOULD be done over a TLS connection.

13. IANA Considerations

There are several IANA considerations associated with this specification.

13.1 XCAP Application Usage IDs

This specification instructs IANA to create a new registry for XCAP application usage IDs (AUIDs). This registry is defined as a table that contains three colums:

AUID: This will be a string provided in the IANA registrations into the registry.

Description: This is text that is supplied by the IANA registration into the registry.

Document: This is a reference to the RFC containing the registration.

This specification instructs IANA to create this table with an initial entry. The resulting table would look like:

Application Unique ID (AUID)	Description	Document
xcap-caps	Capabilities of an XCAP server	RFC XXXX

[[NOTE TO IANA/RFC-EDITOR: Please replace XXXX with the RFC number of this specification.]]

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 [15] characters only.

Descriptive text that describes the application usage.

13.2 MIME Types

This specification requests the registration of several new MIME types according to the procedures of RFC 2048 [7] and guidelines in RFC 3023 [8].

13.2.1 application/xcap-el+xml MIME Type

This specification registers a new MIME type according to the

MIME media type name: application
MIME subtype name: xcap-el+xml
Mandatory parameters: none
Optional parameters: Same as charset parameter application/xml as specified in RFC 3023 [8].
Encoding considerations: Same as encoding considerations of application/xml as specified in RFC 3023 [8].
Security considerations: See Section 10 of RFC 3023 [8].
Interoperability considerations: none.
Published specification: RFC XXXX [[NOTE TO RFC EDITOR: Please replace XXXX with the published RFC number of this specification.]]
Applications which use this media type: This document type has been used to support transport of XML fragment bodies in RFC XXXX [[NOTE TO RFC EDITOR: Please replace XXXX with the published RFC number of this specification.]], the XML Configuration Access Protocol (XCAP).
Additional Information:
Magic Number: None
File Extension: .xel or .xml
Macintosh file type code: "TEXT"
Personal and email address for further information: Jonathan Rosenberg, jdrosen@jdrosen.net
Intended usage: COMMON
Author/Change controller: The IETF.

13.2.2 application/xcap-att+xml MIME Type

MIME media type name: application
MIME subtype name: xcap-att+xml
Mandatory parameters: none
Optional parameters: Same as charset parameter application/xml as specified in RFC 3023 [8].
Encoding considerations: Same as encoding considerations of application/xml as specified in RFC 3023 [8].
Security considerations: See Section 10 of RFC 3023 [8].
Interoperability considerations: none.
Published specification: RFC XXXX [[NOTE TO RFC EDITOR: Please replace XXXX with the published RFC number of this specification.]].
Applications which use this media type: This document type has been used to support transport of XML attribute values in RFC XXXX [[NOTE TO RFC EDITOR: Please replace XXXX with the published RFC number of this specification.]], the XML Configuration Access Protocol (XCAP).
Additional Information:

Magic Number: None
File Extension: .xav
Macintosh file type code: "TEXT"
Personal and email address for further information: Jonathan Rosenberg, jdrosen@jdrosen.net
Intended usage: COMMON
Author/Change controller: The IETF.

13.2.3 application/xcap-error+xml MIME Type

MIME media type name: application
MIME subtype name: xcap-error+xml
Mandatory parameters: none
Optional parameters: Same as charset parameter application/xml as specified in RFC 3023 [8].
Encoding considerations: Same as encoding considerations of application/xml as specified in RFC 3023 [8].
Security considerations: See Section 10 of RFC 3023 [8].
Interoperability considerations: none.
Published specification: This specification.
Applications which use this media type: This document type conveys error conditions defined in RFC XXXX. [[NOTE TO RFC EDITOR: Please replace XXXX with the published RFC number of this specification.]]
Additional Information:
Magic Number: None
File Extension: .xe
Macintosh file type code: "TEXT"
Personal and email address for further information: Jonathan Rosenberg, jdrosen@jdrosen.net
Intended usage: COMMON
Author/Change controller: The IETF.

13.2.4 application/xcap-caps+xml MIME Type

MIME media type name: application
MIME subtype name: xcap-caps+xml
Mandatory parameters: none
Optional parameters: Same as charset parameter application/xml as specified in RFC 3023 [8].
Encoding considerations: Same as encoding considerations of application/xml as specified in RFC 3023 [8].
Security considerations: See Section 10 of RFC 3023 [8].
Interoperability considerations: none.
Published specification: This specification.
Applications which use this media type: This document type conveys capabililites of an XML Configuration Access Protocol (XCAP) server, as defined in RFC XXXX. [[NOTE TO RFC EDITOR: Please replace XXXX with the published RFC number of this specification.]]

Additional Information:
Magic Number: None
File Extension: .xe
Macintosh file type code: "TEXT"
Personal and email address for further information: Jonathan Rosenberg, jdrosen@jdrosen.net
Intended usage: COMMON
Author/Change controller: The IETF.

13.3 URN Sub-Namespace Registrations

This specification registers several new XML namespaces, as per the guidelines in RFC 3688 [16].

13.3.1 urn:ietf:params:xml:ns:xcap-error

URI: The URI for this namespace is urn:ietf:params:xml:ns:xcap-error
Registrant Contact: IETF, SIMPLE working group, (simple@ietf.org), Jonathan Rosenberg (jdrosen@jdrosen.net).
XML:

```
              BEGIN
              <?xml version="1.0"?>
              <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
                        "http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">
              <html xmlns="http://www.w3.org/1999/xhtml">
              <head>
                <meta http-equiv="content-type"
                   content="text/html;charset=iso-8859-1"/>
                <title>XCAP Error Namespace</title>
              </head>
              <body>
                <h1>Namespace for XCAP Error Documents</h1>
                <h2>urn:ietf:params:xml:ns:xcap-error</h2>
                <p>See <a href="[URL of published RFC]">RFCXXXX [[NOTE
TO RFC-EDITOR/IANA: Please replace XXXX with the RFC Number of
this specification]]</a>.</p>
              </body>
              </html>
              END
```

13.3.2 urn:ietf:params:xml:ns:xcap-caps

URI: The URI for this namespace is urn:ietf:params:xml:ns:xcap-caps
Registrant Contact: IETF, SIMPLE working group, (simple@ietf.org), Jonathan Rosenberg (jdrosen@jdrosen.net).

XML:

```
              BEGIN
              <?xml version="1.0"?>
              <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
                        "http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">
              <html xmlns="http://www.w3.org/1999/xhtml">
              <head>
                <meta http-equiv="content-type"
                   content="text/html;charset=iso-8859-1"/>
                <title>XCAP Capabilities Namespace</title>
              </head>
              <body>
                <h1>Namespace for XCAP Capability Documents</h1>
                <h2>urn:ietf:params:xml:ns:xcap-caps</h2>
                <p>See <a href="[URL of published RFC]">RFCXXXX [[NOTE
TO RFC-EDITOR/IANA: Please replace XXXX with the RFC Number of
this specification]]</a>.</p>
              </body>
              </html>
              END
```

13.4 XML Schema Registrations

This section registers two XML schemas per the procedures in [16].

13.4.1 XCAP Error Schema Registration

URI: urn:ietf:params:xml:schema:xcap-error
Registrant Contact: IETF, SIMPLE working group, (simple@ietf.org), Jonathan Rosenberg (jdrosen@jdrosen.net).
XML Schema: The XML for this schema can be found as the sole content of Section 9.2.

13.4.2 XCAP Capabilities Schema Registration

URI: urn:ietf:params:xml:schema:xcap-caps
Registrant Contact: IETF, SIMPLE working group, (simple@ietf.org), Jonathan Rosenberg (jdrosen@jdrosen.net).
XML Schema: The XML for this schema can be found as the sole content of Section 10.2.

14. Acknowledgements

The author would like to thank Ben Campbell, Eva-Maria Leppanen, Hisham Khartabil, Chris Newman, Joel Halpern, Jari Urpalainen, and Lisa Dusseault for their input and comments. A special thanks to Ted Hardie for his input and support.

15. References

15.1 Normative References

[1] Bray, T., Paoli, J., Sperberg-McQueen, C. and E. Maler, "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C FirstEdition REC-xml-20001006, October 2000.

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

[3] Bray, T., Hollander, D. and A. Layman, "Namespaces in XML", W3C REC REC-xml-names-19990114, January 1999.

[4] Grosso, P. and D. Veillard, "XML Fragment Interchange", W3C CR CR-xml-fragment-20010212, February 2001.

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

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

[7] Freed, N., Klensin, J. and J. Postel, "Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures", BCP 13, RFC 2048, November 1996.

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

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

[10] 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.

[11] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

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

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

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

[15] 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.

[16] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, January 2004.

[17] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

15.2 Informative References

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

[19] 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.

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

[21] 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.

[22] 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.

[23] Rosenberg, J., "An Extensible Markup Language (XML) Configuration Access Protocol (XCAP) Usage for Presence Lists", draft-ietf-simple-xcap-list-usage-02 (work in progress), February 2004.

[24] Peterson, J., "Common Profile for Presence (CPP)", draft-ietf-impp-pres-04 (work in progress), October 2003.

[25] Newman, C. and J. Myers, "ACAP -- Application Configuration

Access Protocol", RFC 2244, November 1997.

[26] Day, M., Rosenberg, J. and H. Sugano, "A Model for Presence and Instant Messaging", RFC 2778, February 2000.

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

[28] 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

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

Copyright Statement

Acknowledgment

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