XEP-0075: Jabber Object Access Protocol (JOAP)

Evan Prodromou
mailto:evan@prodromou.san-francisco.ca.us
xmpp:EvanProdromou@jabber.org

2003-05-22
Version 0.3

Status	Type	Short Name
Deferred	Standards Track	N/A

The Jabber Object Access Protocol, or JOAP, defines a mechanism for creating Jabber-accessible object servers, and manipulating objects provided by those servers. It is intended for development of business applications with Jabber.

Legal

Copyright

Permissions

Warranty

Liability

Conformance

Contents

1 Introduction

This document defines the Jabber Object Access Protocol (JOAP) as an extension to the Jabber protocol. It outlines the addressing scheme and IQ stanzas that comprise the protocol as well as the data types that the protocol models. Example applications are discussed, as well as security considerations.
Jabber has a number of attractive features that give it an advantage over existing frameworks for building multi-tier applications, such as the Simple Object Access Protocol (SOAP) or Java 2, Enterprise Edition (J2EE). Among these are:

- **Built-in authentication.** All clients in the Jabber network must be authenticated with their server before sending messages to other Jabber entities. Inter-server communication requires additional authentication, in the form of dialback connections or other trust mechanisms. This ensures that, when a message is delivered to a JOAP object server, there is little doubt as to the authenticity of its originator.
- **Global namespace.** Jabber allows namespacing of addresses according to domain names. This allows objects to be accessed globally, according to authorization rules.
- **An asynchronous messaging model.** Jabber is built on a store-and-forward mechanism that allows a single client to send messages to multiple servers concurrently. Of course, synchronous messaging can be simulated on the client side.
- **Cross-enterprise messaging.** Jabber is designed to allow cross-enterprise messaging. Using Jabber for multitier applications makes development of cross-enterprise systems as easy as intra-enterprise development.
- **Message routing.** The architecture of Jabber is based on clients that connect to a local server, and then can send messages through that server to other clients, servers, or components. Topographically, this contrasts well with the one-to-one connections required with, for example, HTTP.
- **Factoring network connections out of scalability.** Because messages in Jabber are routed, components need to maintain only one connection -- to their upstream Jabber server. This removes the number of network connections from the scalability equation for object servers.
- **Language independence.** Jabber protocol implementations exist for Java, C, and C++ as well as a number of scripting languages such as Perl and Python.
- **Platform independence.** The Jabber protocol is implemented on most major modern platforms.

For existing Jabber development efforts, there are significant advantages to building applications within a JOAP framework. It should go without saying that, for developers creating business applications on top of Jabber, a uniform object access protocol provides significant

advantage for cross-product integration.
In addition, implementers of special-purpose components, such as multi-user chat servers or whiteboarding components, can use an object-server interface to allow fine-grained control of the implementations, especially where such control is not specified by the applicable Jabber protocol.

2 Requirements

JOAP has the following design goals:

- Create a protocol for client programs to access object server components.
- Create a model for addressing object servers, classes, and instances.
- Define a language for manipulating data.
- Define a language for describing data structures.
- Maintain compatibility with Jabber-RPC (XEP-0009) [1].
- Make classes and instances directly addressable.
- Allow human- and programming-language independence.
- Allow easy dynamic mapping of JOAP classes to local classes in Perl, Python, and other dynamic languages.

The following are non-goals:

- Enable object-oriented access to all parts of the Jabber network (e.g., clients, other components).
- Define a language for creating or altering classes on an object server.
- Define a language for describing or defining the authorization of given users for given objects.
- Define a programming interface between object servers and classes in those servers (like Enterprise Java Beans).

[1] XEP-0009: Jabber-RPC <https://xmpp.org/extensions/xep-0009.html>.

3 Overview

The JOAP interface is made up of three key parts:

- A scheme for defining the addresses of object servers, classes, and instances (collectively known as ”objects”).
- A set of message stanzas in the jabber:iq:joap namespace for manipulating data in object servers, classes, and instances. The stanzas allow client programs to analyze the structure of objects, to read object attributes, to edit object attributes, to add new instances, to delete instances, and to search classes.
- An application of XEP-0009 for calling methods on objects.

4 Entities in the JOAP Universe

This section describes the various entities in the JOAP universe. Some entities are directly addressable with Jabber IDs (JIDs), as described below. Others are not considered outside of their enclosing entities.

4.1 Object Server Component

An object server component is a Jabber component that provides object services. It is addressed like any other Jabber component, i.e., with a DNS hostname or pseudo-hostname. Some examples would be:

- payroll.example.com - A payroll application server.
- jukebox.example.com - An MP3 jukebox server.

An object server has zero or more attributes, methods, and classes.

4.2 Class

A class is a category of object instances. It defines the structure and interface of these instances. Each class is addressed using the class name as the node identifier, and the object server as the domain identifier. Class names must conform to the node identifier restrictions defined for XMPP. Class names must also be unique, regardless of case, within an object server. For example:

- Employee@payroll.example.com - An employee class at the payroll.example.com server.

- Song@jukebox.example.net - A song class on the jukebox server.
- Board@circuit-design.example.com - A class for circuit boards.
- Board@surf-shop.example.net - A class for surfboards -- distinct from above class!

Beside uniqueness and XMPP compliance, no further requirements are made on class names. However, good design suggests mnemonic names.
Classes define the attributes and methods of their instances. In addition, they can have attributes and methods of their own. Finally, classes can have superclasses, which indicate an inheritance structure as well as implementation of a defined interface.
JOAP allows for no relative addressing of classes. Classes are always referred to by their full address (node identifier plus domain identifier).

4.3 Instance

An instance is a collection of data with identity, state, and behavior. Each instance is a member of a class, which defines the attributes (data) and methods (behavior) of the instance itself.
An instance is addressed using the node plus server that identifies its class, as well as a unique string that occupies the resource identifier section of the Jabber ID. The resource is only unique over the space of the corresponding class. Some example instance addresses:

- Room@hotel.example.com/103 - Room 103 in the Example Hotel.
- Element@periodic-table.example.net/103 - Element 103 (rutherfordium) in the periodic table.
- Employee@payroll.example.com/JohnSmith - An employee named ”John Smith”.
- Customer@videorental.example.net/JohnSmith - A customer named ”John Smith” (not necessarily the same person as the above employee!).

Besides uniqueness within a class, and compliance with the rules for resource identifiers in the XMPP standard, there are no further requirements on instance identifiers in JOAP. In particular, the instance identifier is opaque -- that is, no further information about the state of the object can or should be discerned from the identifier. What visible part of the instance, if any, makes up the unique resource identifier is implementation dependent.
That said, it is recommended that the instance identifier be persistent through the life of the instance. In addition, using mnemonic identifiers can greatly enhance the usability of JOAP objects.
As with other resource identifiers, instance identifiers are case-sensitive.
The instance identifier roughly corresponds to a primary key in a relational database, and for object servers that provide access to relational databases, it is recommended to use the primary key of a table as the instance identifier. For tables with a compound key, a comma

(',') dash ('-'), or other non-alphanumeric character can be used to separate parts of the key for better readability. For example:

- Date@calendar.example.net/2003-01-26 -- The date January 26th, 2003.
- City@canada.example.com/Montréal,QC -- The city of Montréal, in the province of Québec.

JOAP allows for no relative addressing of instances. Instances are always referred to using their full address (node identifier plus domain identifier plus resource identifier).

4.4 Attribute

An attribute is a unit of state that makes up part of an object server, instance, or class. Each attribute has a name and a type.
Attribute names must be strings of characters containing only the characters [a-zA-Z0-9_]. The first character must be an underscore or alphabetic character. [2]
Attributes cannot be addressed individually. Attributes are manipulated by sending JOAP messages to the object that owns them.

4.5 Method

A method is a unit of behavior that makes up part of an object. Methods in JOAP are compatible with XML-RPC [3], as specified in Jabber-RPC (XEP-0009) [4]. In particular, methods have a name, a return type, and 0 or more parameters, each of which has a type.
The one exception to XML-RPC compatibility is that method names for JOAP are restricted to the characters [a-zA-z0-9_]. [5]
Methods cannot be directly addressed using JOAP. Methods are described and executed by sending messages to the object server, class, or instance that owns them.

5 JOAP Data Types

The range of JOAP data types is borrowed directly from XML-RPC.

[2] This requirement is intended to allow easy mapping of attributes in JOAP to attributes of objects in client programming languages. The restriction is the lowest common denominator for variable names in most modern programming languages.
[3] XML-RPC <http://www.xmlrpc.com/spec>.
[4] XEP-0009: Jabber-RPC <https://xmpp.org/extensions/xep-0009.html>.
[5] This is to avoid conceptual mismatch in programming languages where the other three characters allowed by XML-RPC, namely ".", ":", and "/", are used to separate class or instance names from methods.

5.1 Scalar Types

The scalar types include the following:

- **int** or **i4**: a 32-bit signed integer
- **boolean**: a one-digit integer representing ”true” (1) or ”false” (0)
- **string**: a string of characters
- **double**: double-precision signed floating-point number
- **datetime.iso8601**: a date value, in ISO 8601 format
- **base64**: binary data, base64-encoded for transmission

5.2 Instance Addresses

Instance addresses are a special type of string used for referring to instance objects. They can be passed as parameters to methods, or set as attribute values.
If a value can contain an object instance, its type is the address of a class. The address of any object instance that is an instance of that class, or any of its subclasses, can be used in that value.
For example, if Boxcar@trainset.example.com is a subclass of Car@trainset.example.com, then Boxcar@trainset.example.com/569 can be used as a method parameter, or set as an attribute, where Car@trainset.example.com is the defined type.
Because addresses are used for instance values, all methods involving instances are implicitly pass-by-reference. If a pass-by-value functionality is needed, a struct (see below) should be used instead.
Note that attribute and method param types can use classes and instances from other object servers (that is, with different domain identifiers). For instance, an Employee@payroll.example.com class could have an attribute of type Job@hr.example.com.

5.3 Compound Types

There are two compound types defined in XML-RPC.

5.3.1 Arrays

An *array* is an ordered list of values. An array can contain values of any type, including other compound types.
In JOAP, as with XML-RPC, it is not possible to address, set, or delete elements of an array. To set values in an array, the entire new array must be specified.

5.3.2 Structs

A *struct* is a set of name-value pairs organized into a logical grouping. A struct can contain values of any type, including other compound types.
In JOAP, as with XML-RPC, it is not possible to address, set, or delete elements of a struct. To set values in an struct, the entire new struct must be specified.
Structs are useful mainly for groupings of data that do not have independent identity or behavior. Where an object needs identity or behavior, an instance should be used instead of a struct.

5.4 Specifying Types

Types are specified by a string name of the type. This can be one of the XML-RPC types described above, or a class address.[6]

6 JOAP Stanzas

This section defines the Jabber stanzas that make up the JOAP protocol.
Each stanza is an information query (IQ). Except for method calls, the stanzas are all in the 'jabber:iq:joap' namespace. Each of the following sections describes a stanza in that namespace, herein called a "verb". The verbs allow basic access to object servers, classes, and instances.
Not all verbs can be sent to all JOAP entities. The appropriate JOAP entity a verb should be addressed to is noted under the description of the verb.

6.1 <describe>

The <describe> verb requests the interface -- that is, methods, attributes, and classes -- of a given object server or class. The IQ type is "get".
The <describe> verb is useful for creating wrapper classes in JOAP clients, either at runtime or at compile time. It can also be used for object browsers, or for client programs to ascertain that the interface they assume for an object is still valid.

6.1.1 Targets

<describe> verbs can be sent to object servers, classes, and instances. Each will return different data.

[6] Implementers can determine if a specified type is valid by checking it against a list of the XML-RPC types. If it does not match, it should be checked to see if matches the syntax for a class address (node identifier plus domain identifier). Otherwise, it is not a valid type.

- Object servers return zero or more descriptive texts, zero or more attribute definitions, zero or more method definitions, zero or more class names, and a timestamp.
- Classes return zero or more descriptive texts, zero or more attribute definitions, zero or more method definitions, and a timestamp.
- Instances return the exact results of sending the <describe> method to their class. This is for convenience only; it is preferable to send <describe> to the class directly.

6.1.2 Descriptive Texts

Each object description can contain one or more strings of descriptive text. This is to indicate the purpose and usage of the object in human-readable form.
Multiple descriptions are allowed in the hope that they will be used to describe the attribute in multiple languages (differentiated using the xml:lang attribute).

6.1.3 Attribute Definitions

Attribute definitions have the following parts:

- A name, which is a legal attribute name as described above.
- A type, which is a legal JOAP type as described above.
- A flag indicating if the attribute is an attribute of the class itself, or of individual instances.
- A flag indicating if the attribute is writable.
- A flag indicating if the attribute is required.
- One or more strings of descriptive text, to indicate the purpose and usage of this attribute. Multiple descriptions are allowed in the hope that they will be used to describe the attribute in multiple languages (differentiated using the 'xml:lang' attribute).

The attribute definitions returned to a client should include only attributes the user is authorized to access.

6.1.4 Method Definitions

Method definitions have the following parts:

- A name, which is a legal method name as described above.
- A return type, which is a legal JOAP type as described above.

- A flag indicating if the method is a method of the class itself, or of individual instances.
- Zero or more parameters, each of which has a name, a type, and one or more strings of descriptive text.
- One or more strings of descriptive text, indicating the use and behavior of this method.

The method definitions returned to a client should include only methods the user is authorized to access.

6.1.5 Class References

Classes, in superclass definitions and object server interfaces, are always referred to by their full address.

6.1.6 Timestamps

The timestamp is a date-time value in ISO 8601 format, UTC. The timestamp indicates the last time an interface was changed, if that information is available.

6.1.7 Superclasses

The main point of describing the superclasses a class has is to allow clients to make typing distinctions: that is, to determine if a class presents a given interface, or may be provided as a parameter or attribute in another JOAP call.
The list of superclasses given in a class description is flat, not hierarchical. No provision is made to indicate which of a class's superclasses are superclasses of each other, nor is there any implied precedence order in the order of the classes in the returned description.
In addition, no provision is made to define which superclass actually implements any methods or attributes defined.

6.1.8 Flattening

When a class receives a <describe> verb, it must return all its superclasses, including multiple ancestors. It must as well return all the attributes and methods that it responds to, including those defined in its superclasses. This is called a "flattened" description of the class. [7]

[7] Flattening the class interface reduces the need for making multiple "describe" verb calls just to find the interface for one class.

6.1.9 Examples

The following examples illustrate the use of the <describe> verb. [8]
To describe a server, the JOAP client sends this stanza.

Listing 1: Describing An Object Server

```
<iq type='get'
              id='joap_describe_1'
              from='Client@example.com'
              to='trainset.example.com'>
            <describe xmlns='jabber:iq:joap' />
          </iq>
```

The object server returns this response:

Listing 2: Description of an Object Server

```
<iq type='result'
              id='joap_describe_1'
              from='trainset.example.com'
              to='Client@example.com'>
            <describe xmlns='jabber:iq:joap'>
              <desc xml:lang='en-US'>
                This server provides classes for managing a virtual
                remote train set.
              </desc>
              <attributeDescription writable='true'>
                <name>logLevel</name>
                <type>i4</type>
                <desc xml:lang='en-US'>Verbosity level for access
                    logging.</desc>
              </attributeDescription>
              <methodDescription>
                <name>startLogging</name>
                <returnType>boolean</returnType>
                <desc xml:lang='en-US'>Start logging activity on this
                  server. Returns true for success and false for an
                  error.</desc>
              </methodDescription>
              <methodDescription>
                <name>stopLogging</name>
                <returnType>boolean</returnType>
                <desc xml:lang='en-US'>Stop logging activity on this
                  server. Returns true for success and false for an
                  error.</desc>
              </methodDescription>
```

[8] All extended examples in this document refer to a particular object domain, based on a fictional model train set. A UML description of the object domain is available in Appendix D.

```
            <class>Train@trainset.example.com</class>
            <class>Car@trainset.example.com</class>
            <class>Caboose@trainset.example.com</class>
            <class>Engine@trainset.example.com</class>
            <class>Boxcar@trainset.example.com</class>
            <class>PassengerCar@trainset.example.com</class>
            <class>Building@trainset.example.com</class>
            <class>TrackSegment@trainset.example.com</class>
            <class>Switch@trainset.example.com</class>
            <class>Station@trainset.example.com</class>
            <timestamp>2003-01-07T20:08:13Z</timestamp>
          </describe>
        </iq>
```

To describe the Car@trainset.example.com class, the JOAP client sends this stanza to the class for boxcars.

Listing 3: Describing a Class

```
<iq type='get'
            id='joap_describe_2'
            from='Client@example.com'
            to='Boxcar@trainset.example.com' >
          <describe xmlns='jabber:iq:joap' />
        </iq>
```

The class returns this stanza to the JOAP client.

Listing 4: Description of a Class

```
<iq type='result'
            id='joap_describe_2'
            from='Boxcar@trainset.example.com'
            to='Client@example.com'>
          <describe xmlns='jabber:iq:joap'>
            <desc xml:lang='en-US'>
              A Car in the trainset that can be used to ship cargo.
            </desc>
            <attributeDescription writable='false' required='true'>
              <name>trackingNumber</name>
              <type>i4</type>
              <desc xml:lang='en-US'>Tracking number for this car.</
                  desc>
            </attributeDescription>
            <attributeDescription writable='true' required='true'>
              <name>contents</name>
              <type>string</type>
              <desc xml:lang='en-US'>Contents of the boxcar.</desc>
            </attributeDescription>
```

```
            <methodDescription allocation='class'>
              <name>nextTrackingNumber</name>
              <returnType>i4</returnType>
              <desc xml:lang='en-US'>The next available tracking
                  number.</desc>
            </methodDescription>
            <superclass>Car@trainset.example.com</superclass>
            <timestamp>2003-01-07T20:08:13Z</timestamp>
          </describe>
        </iq>
```

To describe an instance, the JOAP client sends this stanza to a particular track segment.

Listing 5: Describing an Instance

```
<iq type='get'
            id='joap_describe_3'
            from='Client@example.com'
            to='TrackSegment@trainset.example.com/134' >
          <describe xmlns='jabber:iq:joap' />
        </iq>
```

The instance returns this stanza to the JOAP client.

Listing 6: Description of an Instance

```
<iq type='result'
            from='TrackSegment@trainset.example.com/134'
            to='Client@example.com'
            id='joap_describe_3'>
          <describe xmlns='jabber:iq:joap'>
            <desc xml:lang='en-US'>
              A length of track in the trainset which can be
              connected to a previous and next length of track.
            </desc>
            <attributeDescription>
              <name>previous</name>
              <type>TrackSegment@trainset.example.com</type>
              <desc>Previous segment of track.</desc>
            </attributeDescription>
            <attributeDescription>
              <name>next</name>
              <type>TrackSegment@trainset.example.com</type>
              <desc>Next segment of track.</desc>
            </attributeDescription>
            <timestamp>2003-01-07T20:08:13Z</timestamp>
          </describe>
        </iq>
```

6.2 <read>

The <read> verb allows clients to retrieve the values of attributes of an object server, class, or instance. The client can specify which attributes to return; if no attributes are specified, then all attributes are returned. [9]
The <read> verb uses the ”get” IQ type.

6.2.1 Timestamps

A timestamp, in ISO 8601 format, UTC, can be added to the results of a <read>. The timestamp indicates the last time any of an object’s attribute values have changed (not just the requested ones). The timestamp can be used, for example, to implement object caching on the client side.

6.2.2 Error Codes

The following are some common error codes may be generated in response to a <read> verb.

- **404 (Not Found)**: The object addressed does not exists.
- **403 (Forbidden)**: The user is not authorized to read attributes of this object, or not authorized to read the specified attributes of this object.
- **406 (Not Acceptable)**: The client sent an <read> verb specifying attributes that are not defined for the class.

6.2.3 Examples

This section gives some examples of using the <read> verb.
A client would send the following stanza to an instance to read its attributes:

Listing 7: Reading the Attributes of an Instance

```
<iq type='get'
              id='joap_read_1'
              from='Client@example.com'
              to='Station@trainset.example.com/Paddington'>
            <read xmlns='jabber:iq:joap' />
          </iq>
```

In return, the instance would send this stanza to the client:

[9] This allows clients to cheaply retrieve meta-information about an instance that may have exceptionally large data, such as bin64-encoded file data.

Listing 8: Attributes of an Instance

```
<iq type='result'
              id='joap_read_1'
              from='Station@trainset.example.com/Paddington'
              to='Client@example.com'>
            <read xmlns='jabber:iq:joap'>
              <attribute>
                <name>name</name>
                <value>Paddington Station</value>
              </attribute>
              <attribute>
                <name>size</name>
                <value>
                  <struct>
                    <member>
                      <name>length</name>
                      <value><i4>4</i4></value>
                    </member>
                    <member>
                      <name>width</name>
                      <value><i4>3</i4></value>
                    </member>
                  </struct>
                </value>
              </attribute>
              <attribute>
                <name>previous</name>
                <value>TrackSegment@trainset.example.com/334</value>
              </attribute>
              <attribute>
                <name>next</name>
                <value>TrackSegment@trainset.example.com/271</value>
              </attribute>
            </read>
          </iq>
```

To read only specified attributes of an instance, the client would send this stanza:

Listing 9: Reading Limited Attributes

```
<iq type='get'
              id='joap_read_2'
              from='Client@example.com'
              to='Train@trainset.example.com/38'>
            <read xmlns='jabber:iq:joap'>
              <name>location</name>
              <name>cars</name>
            </read>
          </iq>
```

In return, the instance would send this stanza to the client:

Listing 10: Limited Attributes

```
<iq type='result'
              id='joap_read_2'
              from='Train@trainset.example.com/38'
              to='Client@example.com'>
            <read xmlns='jabber:iq:joap'>
              <attribute>
                <name>location</name>
                <value>Station@trainset.example.com/Paddington</value>
              </attribute>
              <attribute>
                <name>cars</name>
                <value>
                  <array>
                    <data>
                      <value>Engine@trainset.example.com/14</value>
                      <value>PassengerCar@trainset.example.com/112</
                          value>
                      <value>PassengerCar@trainset.example.com/309</
                          value>
                      <value>BoxCar@trainset.example.com/212</value>
                      <value>Caboose@trainset.example.com/9</value>
                    </data>
                  </array>
                </value>
              </attribute>
            </read>
          </iq>
```

6.3 <add>

The <add> verb is used to create a new instance of a JOAP class. The verb is sent to the JOAP class, which returns the address of the newly-created instance.
Within each <add> verb the client must include attribute values for each required, writable attribute of the class.
The IQ is of type ”set”.

6.3.1 Error Codes

The following are some common error codes may be generated in response to an <add> verb.

- **404 (Not Found)**: The class for which an instance is to be instantiated does not exists.

- **403 (Forbidden)**: The user is not authorized to instantiate an instance of this class.
- **405 (Not Allowed)**: The client sent an <add> verb to something that isn't a class.
- **406 (Not Acceptable)**: The client sent an <add> verb containing attributes that are not writable, or without all required, writable attributes, or with attributes that are not defined for the class, or with attribute values that are of the wrong type.

6.3.2 Examples

To create a new PassengerCar, the client would send the following stanza to the PassengerCar class:

Listing 11: Adding a New Instance

```
<iq type='set'
              id='joap_add_1'
              to='PassengerCar@trainset.example.com'
              from='Client@example.com'>
            <add xmlns='jabber:iq:joap'>
              <attribute>
                <name>passengers</name>
                <value><i4>38</i4></value>
              </attribute>
            </add>
          </iq>
```

The class would return the following response:

Listing 12: A New Instance

```
<iq type='result'
              id='joap_add_1'
              from='PassengerCar@trainset.example.com'
              to='Client@example.com'>
            <add xmlns='jabber:iq:joap'>
              <newAddress>PassengerCar@trainset.example.com/866</
                  newAddress>
            </add>
          </iq>
```

Note that the class created a new instance identifier, 866, for the new instance. Further communications from the client would use the full instance address returned.

6.4 <edit>

The <edit> verb is used to update the attributes of an object. The name and new value of each attribute that is to be updated is listed in the <edit> verb.

The IQ is of type ”set”.
Leaving a given attribute out of an <edit> verb does not indicate that the attribute should be set to an undefined or default value. The new values of attributes that are left out is implementation-dependent; in general, though, they should remain unchanged, if possible.

6.4.1 Content in <edit> Results

If the results of an <edit> verb have content, it will contain the new address of the instance that was updated. The new address should be used henceforth by the client. [10]

6.4.2 Error Codes

The following error codes may be generated in response to a <edit> verb.

- **404 (Not Found)**: The object to be edited does not exists.
- **403 (Forbidden)**: The user is not authorized to edit this object, or to change one of the attributes specified in the <edit> request.
- **406 (Not Acceptable)**: The client sent an <edit> verb containing attributes that are not defined for the class, or with attribute values that are of the wrong type, or with attribute values that are outside the range for the attribute.

6.4.3 Examples

To change the number of passengers in a PassengerCar, the client would send the following stanza to the instance:

Listing 13: Editing an Instance

```
<iq type='set'
              id='joap_edit_1'
              from='Client@example.com'
              to='PassengerCar@trainset.example.com/199'>
            <edit xmlns='jabber:iq:joap'>
              <attribute>
                <name>passengers</name>
                <value><i4>31</i4></value>
              </attribute>
            </edit>
          </iq>
```

[10] This is to allow updates that alter the unique key or attribute of an instance that determine its instance identifier.

The client would return the following stanza:

Listing 14: Results of Editing an Instance

```
<iq type='result'
              id='joap_edit_1'
              to='Client@example.com'
              from='PassengerCar@trainset.example.com/199'>
            <edit xmlns='jabber:iq:joap' />
          </iq>
```

If a client wanted to change the name of a Building, it would send the following stanza to the instance:

Listing 15: Editing an Instance

```
<iq type='set'
              id='joap_edit_2'
              from='Client@example.com'
              to='Building@trainset.example.com/JonesFamilyHome'>
            <edit xmlns='jabber:iq:joap'>
              <attribute>
                <name>name</name>
                <value>Smith Family Home</value>
              </attribute>
            </edit>
          </iq>
```

The results would be as follows:

Listing 16: Results of Editing an Instance

```
<iq type='result'
              id='joap_edit_2'
              to='Client@example.com'
              from='Building@trainset.example.com/JonesFamilyHome'>
            <edit xmlns='jabber:iq:joap'>
              <newAddress>Building@trainset.example.com/
                  SmithFamilyHome</newAddress>
            </edit>
          </iq>
```

Note that the instance indentifier, and thus the instance address, of the instance has changed. The from part of the IQ, however, contains the old address.

6.5 <delete>

The <delete> verb is used to delete an instance. The IQ is of type ”set”. The <delete> stanza has no sub-elements.
Only instances can be deleted. Classes and object servers cannot be deleted. After an instance is deleted, it is no longer addressable.
A given user may not be able to delete a particular instance.

6.5.1 Error Codes

The following error codes may be generated in response to a <delete> verb.

- **404 (Not Found)**: The instance to be deleted does not exists.
- **403 (Forbidden)**: The user is not authorized to delete this instance.
- **405 (Not Allowed)**: The client sent a <delete> verb to an object server or class.

6.5.2 Examples

To delete an instance, a client would send the following stanza:

Listing 17: Deleting an Instance

```
<iq type='set'
              id='joap_delete_1'
              from='Client@example.com'
              to='Building@trainset.example.com/Courthouse'>
            <delete xmlns='jabber:iq:joap' />
          </iq>
```

The instance would return this stanza:

Listing 18: A Deleted Instance

```
<iq type='result'
              id='joap_delete_1'
              to='Client@example.com'
              from='Building@trainset.example.com/Courthouse'>
            <delete xmlns='jabber:iq:joap' />
          </iq>
```

If the user is not authorized to delete the instance, it would return this error:

Listing 19: Error on Unauthorized Deletion

```
<iq type='error'
              id='joap_delete_1'
              to='Client@example.com'
              from='Building@trainset.example.com/Courthouse'>
            <delete xmlns='jabber:iq:joap' />
            <error code='403'>
              You are not authorized to delete this instance.
            </error>
          </iq>
```

6.6 <search>

The <search> verb allows rudimentary searching and listing of instances in a class. The IQ is of type "get".
The client sends a <search> verb to the class, specifying the attributes that are search criteria and values to search for. The class returns a list of the addresses of matching instances.
Multiple attributes are logically AND'd; that is, resulting instances must match *all* of the attribute values.

6.6.1 Value Matching

How attribute values are specified for matching depends on the type of the attribute.

- For numeric types (<int>, <double>), <boolean>, and <dateTime.iso8601>, values match if they are exactly equal.
- For <string> types, a search value matches an attribute value if it is a case-dependent substring of that value. For example, "hat" will match "hat", "that", and "real-time chat server".
- For the <base64> type, a search value matches an attribute value if the base64-decoded value of the search value is an 8-bit clean substring of the base64-decoded attribute value. For example, "aGF0Cg==" ("hat") will match "cmVhbC10aW1lIGNoYXQK" ("real-time chat").
- For instance addresses, a search value matches an attribute value if they are exactly equal.
- For <struct> types, a search value matches an attribute value if each of its named members matches the corresponding named members in the attribute value, and has the same type.

- For <array> types, a search value matches an attribute value if each of its members matches the corresponding members in the attribute value, in order, and has the same type.

6.6.2 Instances of Subclasses

Classes should return all instances of the class that are on the same object server (that is, which have the same domain identifier in their address) and that match the search criteria. This includes instances of subclasses of the class.
Whether a class returns instances of subclasses that reside on other object servers is implementation-dependent. [11]
Classes cannot be searched on attributes that are defined only in subclasses; for example, a search for the attribute ”contents” sent to the Car@trainset.example.com class should result in a 406 (Not Acceptable) error.

6.6.3 Empty <search>

The semantics of an empty <search> verb is to request *all* instances of a class. This provides a listing or browsing functionality.

6.6.4 Error Codes

The following error codes may be generated in response to a <search> verb.

- **404 (Not Found)**: The class to be searched does not exists.
- **403 (Forbidden)**: The user is not authorized to search this class.
- **405 (Not Allowed)**: The client sent a <search> verb to an object server or instance.
- **406 (Not Acceptable)**: The client sent an <search> verb containing attributes that are not defined for the class, or with attribute values that are of the wrong type.

6.6.5 Examples

To search for Boxcar instances carrying coal, the client would send the following stanza to the Boxcar class:

[11] This caveat is to allow different types of subclassing policies. Classes that define a well-known, standard interface -- for example, a class defined by a standards organization -- would probably not be ”aware” of all instances of that class. However, it is conceivable to have a multi-tier business application where the object servers did know about other servers, their classes, and their instances.

Listing 20: Searching for Instances

```
<iq type='get'
              id='joap_search_1'
              from='Client@example.com'
              to='Boxcar@trainset.example.com'>
            <search xmlns='jabber:iq:joap'>
              <attribute>
                <name>contents</name>
                <value><string>coal</string></value>
              </attribute>
            </search>
          </iq>
```

The Boxcar class would return a list of all matching instances:

Listing 21: Search Results

```
<iq type='result'
              id='joap_search_1'
              from='Boxcar@trainset.example.com'
              to='Client@example.com'>
            <search xmlns='jabber:iq:joap'>
              <item>Boxcar@trainset.example.com/195</item>
              <item>Boxcar@trainset.example.com/35</item>
              <item>Boxcar@trainset.example.com/681</item>
            </search>
          </iq>
```

To get a list of all Building instances, the client would send an empty <search> verb, as follows:

Listing 22: Listing All Instances of a Class

```
<iq type='get'
              id='joap_search_2'
              from='Client@example.com'
              to='Building@trainset.example.com'>
            <search xmlns='jabber:iq:joap' />
          </iq>
```

The Building class would return the following stanza:

Listing 23: List Results

```
<iq type='result'
              id='joap_search_2'
              from='Building@trainset.example.com'
              to='Client@example.com'>
            <search xmlns='jabber:iq:joap'>
              <item>Building@trainset.example.com/Courthouse</item>
```

```
            <item>Station@trainset.example.com/Paddington</item>
            <item>Station@trainset.example.com/GareDeLyon</item>
            <item>Building@trainset.example.com/SmithFamilyHome</
                item>
          </search>
        </iq>
```

Note that the class returns instances of subclasses, as well as direct instances of the class.

6.7 Method Calls

Method calls in JOAP are simply XML-RPC calls, as defined in XEP-0009.[12] To call a method on an object, the client simply sends an XML-RPC message to that object. Method calls must match the parameters as defined in the method definition returned by the <describe> verb. Method names must be the exact method name as returned by <describe>. No class or instance identifier prefix (with "." or ":") is used.
Note, also, that the addressee of the method call, that is, the object that defines the method, is not specified as a parameter of the method, as it is in some programming languages. The addressee of the method is implicit in the address to which the method was sent.

6.7.1 Examples

To start the event log on the train set server, the client would send the following stanza:

Listing 24: Method Call on an Object Server

```
<iq type='set'
            id='joap_xmlrpc_1'
            from='Client@example.com'
            to='trainset.example.com'>
          <query xmlns='jabber:iq:rpc'>
            <methodCall>
              <methodName>startLogging</methodName>
            </methodCall>
          </query>
        </iq>
```

The object server would respond with the following results:

Listing 25: Method Call on an Object Server

```
<iq type='result'
            id='joap_xmlrpc_1'
```

[12] XEP-0009 leaves some open questions as to use of widely-defined extensions to the XML-RPC standard, such as the <nil> type.

```
            to='Client@example.com'
            from='trainset.example.com'>
          <query xmlns='jabber:iq:rpc'>
            <methodResponse>
              <params>
                <param>
                  <value><boolean>1</boolean></value>
                </param>
              </params>
            </methodResponse>
          </query>
        </iq>
```

To retrieve the next available Car tracking number, the client would send the following stanza to the Car class:

Listing 26: Method Call on a Class

```
<iq type='set'
            id='joap_xmlrpc_2'
            from='Client@example.com'
            to='Car@trainset.example.com'>
          <query xmlns='jabber:iq:rpc'>
            <methodCall>
              <methodName>nextTrackingNumber</methodName>
            </methodCall>
          </query>
        </iq>
```

The class would respond with the following results:

Listing 27: Results of a Class Method Call

```
<iq type='result'
            id='joap_xmlrpc_2'
            to='Client@example.com'
            from='Car@trainset.example.com'>
          <query xmlns='jabber:iq:rpc'>
            <methodResponse>
              <params>
                <param>
                  <value><i4>909</i4></value>
                </param>
              </params>
            </methodResponse>
          </query>
        </iq>
```

To make a Switch change to a different track segment, the client would send the following stanza to the instance:

Listing 28: Method Call on an Instance

```
<iq type='set'
              id='joap_xmlrpc_3'
              from='Client@example.com'
              to='Switch@trainset.example.com/981'>
            <query xmlns='jabber:iq:rpc'>
              <methodCall>
                <methodName>switchTo</methodName>
                <params>
                  <param>
                    <value>TrackSegment@trainset.example.com/119</
                        value>
                  </param>
                </params>
              </methodCall>
            </query>
          </iq>
```

The instance would respond with the following results:

Listing 29: Results of an Instance Method Call

```
<iq type='result'
              id='joap_xmlrpc_3'
              from='Switch@trainset.example.com/981'
              to='Client@example.com'>
            <query xmlns='jabber:iq:rpc'>
              <methodResponse>
                <params>
                  <param>
                    <value><boolean>1</boolean></value>
                  </param>
                </params>
              </methodResponse>
            </query>
          </iq>
```

7 Potential Applications

7.1 Application Server

A simple application server can be provided using JOAP. This is merely the degenerate case of an object server that provides only methods and attributes, with no classes.

7.2 Relational Database Interface

A more complex example would be an interface to a relational database server, such as Oracle, PostgreSQL, or mySQL. The object server would represent a single database within the database server. Each table in the database would be represented by a class with no class attributes or methods. Each row in the database would be an instance of its table's class, with attributes but no methods.

7.3 N-Tier Application

A distributed n-tier application can be built fairly directly with JOAP. N-tier applications are usually defined as having three main segments:

- A user-interface segment
- A business-object segment, defining objects with business rules encoded into their behavior
- A data-storage segment, handling basic storage of relatively unintelligent objects

With JOAP, application developers can create the last two segments with a JOAP interface. User-interface clients can use JOAP to access and manipulate the business objects in a business object server. In turn, the business objects can use JOAP to manipulate underlying database objects in the data storage layer (perhaps implemented using a relational database interface, as defined above).

7.4 Jabber Component Controller

Jabber protocols typically define a base set of functionality for a component to provide. Implementers often want to provide specialized, fine-grained control of the component that is not part of the core functionality of a component. For example, the implementer may wish to allow administrators to get metrics on a component, enable or review logs, note error situations, or configure the component remotely.[13]
A component can provide an additional JOAP interface, along with its regular protocol-specific interface, to enable this kind of control functionality. Implementers can in this way provide implementation-specific functionality in an open way.
For example, if conference.example.com is a MUC component, control.conference.example.com might be a JOAP component with access to the internal data structures of the MUC component. A conference room addressed in the MUC component as ModelTrains@conference.example.com might be addressed in the JOAP component as Room@control.conference.example.com/ModelTrains.

[13] Most Jabber components currently define Web interfaces, or command-line scripts, to perform this kind of control.

7.5 Distributed Object System Gateway

There are a number of existing distributed object systems, such as SOAP, CORBA, distributed OLE, Enterprise Java Beans, etc.
It would be reasonable to create gateways for these object systems or object servers implementing their protocols using JOAP. JOAP could also be used to allow disparate object systems to communicate through a common protocol.

8 Implementation Notes

To follow.

9 Security Considerations

This section describes some security considerations for implementers of JOAP.

9.1 Authentication

No provision is made for authentication of users to the object server. Jabber users authenticate to a login server before they are able to send any Jabber stanzas.

9.2 Authorization

Authorization for users to access and manipulate objects and attributes in JOAP is fine-grained; object servers can return error codes to indicate a lack of authorization for any given attribute, object, or method.
No provision is made to define a user's authorization for an object, attribute, or method. Implicit authorization is outlined with the results of the <describe> verb.

- For attributes, if a user is unauthorized to <read> the attribute, the object server should not return a definition of the attribute in the <describe> results.
- If a user is unauthorized to <edit> an attribute, the object server should note that the the attribute is not writable in the <describe> results.
- If a user is unauthorized to execute a method, the object server should not return a definition of the attribute in the <describe> results.
- For classes that the user is not allowed to access at all, the object server should not return a reference to that class in the <describe> results for the object server.

- For instances that the user is not allowed to access at all, the object server should not return references to that instance in <search> results.

9.3 Privacy and Confidentiality

No provision is made in the JOAP protocol for providing privacy and confidentiality in JOAP conversations. This is left up to existing, more general Jabber protocols and extensions.
Confidentiality from external, non-Jabber observers can be obtained using transport-layer security (TLS) in all legs of the Jabber path -- from client to server to (potentially) another server to the object server component.
Maintaining confidentiality against observers in the Jabber pathway -- for example, servers relaying JOAP stanzas -- requires using end-to-end encryption.
Due to the nature of the JOAP addressing scheme, however, perfect confidentiality cannot be preserved. Even if the contents of an IQ packet are encrypted, the address of the object the packet is sent to -- e.g., Tips@whistleblower.example.org/NuclearRegulatoryInfractions -- will reveal some information about the JOAP conversation which could be harmful to the user.

10 IANA Considerations

This document requires no interaction with the IANA.

11 XMPP Registrar Considerations

This protocol defines one new namespace, 'jabber:iq:joap'.
Experimental implementations of this protocol should use the namespace 'http://www.xmpp.org/extensions/xep-0075.html#0.3' to avoid conflicts with future versions.

12 Future Considerations

- The presence mechanism provided by Jabber is currently not integrated into the JOAP protocol. It would be relatively easy to incorporate an observation mechanism, based on presence, that would enable JOAP clients to request state updates on JOAP instances. It is currently an open question as to whether this would be genuinely useful, or merely a "gee-whiz" add-on that needlessly complicates implementation of the JOAP protocol.
- More sophisticated distributed object protocol mechanisms, such as transactions, are not addressed in this specification.

- The JOAP addressing mechanism restricts the type of Jabber entity that can act as an object server. A Jabber instant messaging client, for example, is normally addressed with username and resource, such as 'AJabberUser@example.com/Home'. This address has no place to hang the class and instance identifiers to allow JOAP interactions.
- Additionally, the JOAP addressing mechanism inhibits letting components that already use the node and resource identifier parts of their addresses, such as multi-user chat (MUC) services. This restriction can be worked around by providing corresponding JOAP components for existing Jabber components.
- An additional type for object references may be called for.
- The addition of the widely used <nil> XML-RPC data type may be called for.
- Searching is fairly rudimentary; a full boolean logic (and, or, not) may be necessary to provide rich searching.
- There is no indication in class descriptions of the inheritance hierarchy of the class, or which superclass's implementation of a method the class may use. This is an implementation decision, and it seems counterproductive to force any given inheritance method on the implementer.
- Some functionality will be expensive using JOAP. An example might be finding all Cars in a Train and adding them to another Train. In our example, this would require one IQ to get the train's cars, and N many other IQs to update each Car. A batching mechanism -- being able to send multiple updates in one IQ message -- would make this somewhat less chatty. However, this functionality may be better addressed by a global Jabber batching mechanism, rather than special-purpose batching just for JOAP.
- A more appropriate term than <verb> may be necessary to describe the different types of IQs that make up the JOAP namespace.
- The names of the verbs come from the so-called basic READ data-manipulation functions: read, edit, add, delete. A more SQL-oriented set of names might be SELECT, UPDATE, INSERT, DELETE. A set of names from C-related object languages would be get, set, new, destroy or delete.
- No mechanism is provided to describe the range of an attribute. For example, if an attribute can only be an integer less than 10, or a string in the set {'open', 'closed'}, there is no allowance for describing this in JOAP.
- Currently, struct and array attributes and parameters are described just with the "struct" and "array" types, and no definition is given of their parts' types, names, or semantics. It may be valuable to expand JOAP type descriptions to also describe the members of a struct or array.

13 Appendix A: Glossary

The following glossary collects some definitions of terms used in this document.

Object services Modelling an object or collection of objects, and providing an interface to manipulate those objects to other entities.

Object server A Jabber component that provides object services.

Class A category of object instances that defines their structure and interface.

Instance A collection of data with identity (address), state (attributes), and behavior (methods).

Attribute A unit of state that makes up part of an object server, instance, or class.

Method A unit of behavior.

Object An object server, class, or instance.

User A person or process that accesses object services through JOAP.

Client The software or agent a user employs to access object services through JOAP.

Instance address The full JID of an instance, e.g., Train@trainset.example.com/OrangeBlossomSpecial.

Instance identifier The resource identifier part of an instance address. For example, in Train@trainset.example.com/OrangeBlossomSpecial, the instance identifier is OrangeBlossomSpecial.

Class address The full JID of a class, e.g., Switch@trainset.example.com.

Class identifier The node identifier part of a class address. For example, in Switch@trainset.example.com, the class identifier is Switch

Authentication The act of determining that a user is who they say they are. In the Jabber world, this is done at login time.

Authorization The act of determining whether a given user has the right to execute a particular action.

14 Appendix B: JOAP XML Schema

The following is an XML Schema for JOAP.

```
<?xml version='1.0' encoding='UTF-8'?>
<schema xmlns='http://www.w3.org/2001/XMLSchema'
  xmlns:joap='jabber:iq:joap'
  targetNamespace='jabber:iq:joap'
  elementFormDefault='qualified'
  attributeFormDefault='unqualified'>
  <element name='describe'>
    <complexType>
      <choice>
        <group ref='joap:DescribeRequest' />
        <group ref='joap:DescribeResponse' />
      </choice>
    </complexType>
  </element>
  <element name='read'>
    <complexType>
      <choice>
        <group ref='joap:ReadRequest' />
        <group ref='joap:ReadResponse' />
      </choice>
    </complexType>
  </element>
  <element name='edit'>
    <complexType>
      <choice>
        <group ref='joap:EditRequest' />
        <group ref='joap:EditResponse' />
      </choice>
    </complexType>
  </element>
  <element name='add'>
    <complexType>
      <choice>
        <group ref='joap:AddRequest' />
        <group ref='joap:AddResponse' />
      </choice>
    </complexType>
  </element>
  <element name='delete'>
    <complexType>
      <choice>
        <group ref='joap:DeleteRequest' />
        <group ref='joap:DeleteResponse' />
      </choice>
    </complexType>
  </element>
  <element name='search'>
    <complexType>
      <choice>
```

```
      <group ref='joap:SearchRequest' />
      <group ref='joap:SearchResponse' />
    </choice>
  </complexType>
</element>
<group name='DescribeRequest'>
  <sequence /> <!-- empty -->
</group>
<group name='DescribeResponse'>
  <sequence>
    <element name='desc' type='joap:Description'
      minOccurs='0' maxOccurs='unbounded' />
    <element name='attributeDescription' type='
       joap:AttributeDescription'
      minOccurs='0' maxOccurs='unbounded' />
    <element name='methodDescription' type='joap:MethodDescription'
      minOccurs='0' maxOccurs='unbounded' />
    <choice>
      <element name='superclass' type='joap:ClassAddress'
        minOccurs='0' maxOccurs='unbounded' />
      <element name='class' type='joap:ClassAddress'
        minOccurs='0' maxOccurs='unbounded' />
    </choice>
    <element name='timestamp' type='joap:Timestamp'
      minOccurs='0' maxOccurs='1' />
  </sequence>
</group>
<group name='ReadRequest'>
  <sequence>
    <element name='name' type='joap:JOAPName'
      minOccurs='0' maxOccurs='unbounded' />
  </sequence>
</group>
<group name='ReadResponse'>
  <sequence>
    <element name='attribute' type='joap:Attribute'
      minOccurs='0' maxOccurs='unbounded' />
    <element name='timestamp' type='joap:Timestamp'
      minOccurs='0' maxOccurs='1' />
  </sequence>
</group>
<group name='EditRequest'>
  <sequence>
    <element name='attribute' type='joap:Attribute'
      minOccurs='0' maxOccurs='unbounded' />
  </sequence>
</group>
<group name='EditResponse'>
  <sequence>
```

```
    <element name='newAddress' type='joap:InstanceAddress'
      minOccurs='0' maxOccurs='1' />
  </sequence>
</group>
<group name='AddRequest'>
  <sequence>
    <element name='attribute' type='joap:Attribute'
      minOccurs='0' maxOccurs='unbounded' />
  </sequence>
</group>
<group name='AddResponse'>
  <sequence>
    <element name='newAddress' type='joap:InstanceAddress'
      minOccurs='1' maxOccurs='1' />
  </sequence>
</group>
<group name='DeleteRequest'>
  <sequence /> <!-- empty -->
</group>
<group name='DeleteResponse'>
  <sequence /> <!-- empty -->
</group>
<group name='SearchRequest'>
  <sequence>
    <element name='attribute' type='joap:Attribute'
      minOccurs='0' maxOccurs='unbounded' />
  </sequence>
</group>
<group name='SearchResponse'>
  <sequence>
    <element name='item' type='joap:InstanceAddress'
      minOccurs='0' maxOccurs='unbounded' />
  </sequence>
</group>
<complexType name='Attribute'>
  <sequence>
      <element name='name' type='joap:JOAPName'
      minOccurs='1' maxOccurs='1' />
      <element name='value' type='joap:JOAPValue'
      minOccurs='1' maxOccurs='1' />
  </sequence>
</complexType>
<complexType name='AttributeDescription'>
  <!-- XXX: enforce name rules -->
  <sequence>
    <element name='name' type='joap:JOAPName'
      minOccurs='1' maxOccurs='1' />
    <element name='type' type='joap:JOAPType'
      minOccurs='1' maxOccurs='1' />
```

```
    <element name='desc' type='joap:Description'
      minOccurs='0' maxOccurs='unbounded' />
  </sequence>
  <attribute name='required' type='boolean' default='false' />
  <attribute name='writable' type='boolean' default='false' />
  <attribute name='allocation' type='joap:Allocation' />
</complexType>
<complexType name='MethodDescription'>
  <sequence>
    <element name='name' type='joap:JOAPName'
      minOccurs='1' maxOccurs='1' />
    <element name='returnType' type='joap:JOAPType'
      minOccurs='1' maxOccurs='1'/>
    <element name='params' minOccurs='0' maxOccurs='1'>
      <complexType>
        <sequence>
          <element name='param' minOccurs='0' maxOccurs='unbounded'
            >
            <complexType>
              <sequence>
                <element name='name' type='joap:JOAPName'
                  minOccurs='1' maxOccurs='1' />
                <element name='type' type='joap:JOAPType'
                  minOccurs='1' maxOccurs='1' />
                <element name='desc' type='joap:Description'
                  minOccurs='0' maxOccurs='unbounded' />
              </sequence>
            </complexType>
          </element>
        </sequence>
      </complexType>
    </element>
    <element name='desc' type='joap:Description'
       minOccurs='0' maxOccurs='unbounded' />
  </sequence>
  <attribute name='allocation' type='joap:Allocation' />
</complexType>
<simpleType name='ClassAddress'>
  <restriction base='string'>
    <pattern value='[^@]+@[a-zA-Z0-9\.]+' />
  </restriction>
</simpleType>
<simpleType name='InstanceAddress'>
  <restriction base='string'>
    <pattern value='[^@]+@[a-zA-Z0-9\.]+/.+' />
  </restriction>
</simpleType>
<simpleType name='XMLRPCType'>
  <restriction base='string'>
```

```
      <enumeration value='int' />
      <enumeration value='i4' />
      <enumeration value='double' />
      <enumeration value='boolean' />
      <enumeration value='string' />
      <enumeration value='array' />
      <enumeration value='struct' />
    </restriction>
  </simpleType>
  <simpleType name='JOAPType'>
    <union memberTypes='ClassAddress XMLRPCType' />
  </simpleType>
  <simpleType name='JOAPName'>
    <restriction base='string'>
      <pattern value='[a-zA-Z_][0-9a-zA-Z_]*'/>
    </restriction>
  </simpleType>
  <simpleType name='Boolean'>
    <restriction base='unsignedByte'>
      <enumeration value='0' />
      <enumeration value='1' />
    </restriction>
  </simpleType>
  <!-- FIXME: figure out how to do this without using mixed
  content, which allows stuff we don't want. -->
  <complexType name='JOAPValue' mixed='true'>
    <choice minOccurs='0'>
      <element name='i4' type='integer' />
      <element name='int' type='integer' />
      <element name='boolean' type='joap:Boolean' />
      <element name='string' type='string' />
      <element name='double' type='decimal' />
      <element name='datetime.iso8601' type='dateTime' />
      <element name='base64' type='base64Binary' />
      <element name='struct' type='joap:XMLRPCStruct' />
      <element name='array' type='joap:XMLRPCArray' />
    </choice>
  </complexType>
  <complexType name='XMLRPCStruct'>
    <sequence>
      <element name='member' minOccurs='1' maxOccurs='unbounded'>
        <complexType>
          <sequence>
            <element name='name' minOccurs='1' maxOccurs='1'
                type='string' />
            <element name='value' minOccurs='1' maxOccurs='1'
                type='joap:JOAPValue' />
          </sequence>
        </complexType>
```

```
      </element>
    </sequence>
  </complexType>
  <complexType name='XMLRPCArray'>
    <sequence>
      <element name='data' minOccurs='1' maxOccurs='1'>
        <complexType>
          <sequence>
            <element name='value' type='joap:JOAPValue'
              minOccurs='0' maxOccurs='unbounded' />
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
  <simpleType name='Allocation'>
    <restriction base='string'>
      <enumeration value='instance' />
      <enumeration value='class' />
    </restriction>
  </simpleType>
  <simpleType name='Description'>
    <restriction base='string' />
  </simpleType>
  <simpleType name='Timestamp'>
    <restriction base='dateTime' />
  </simpleType>
</schema>
```

15 Appendix C: JOAP DTD

The following is a document-type description (DTD) for JOAP.

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT desc (#PCDATA)>
<!ATTLIST desc
          xml:lang NMTOKEN #IMPLIED>
<!ELEMENT i4 (#PCDATA)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT double (#PCDATA)>
<!ELEMENT datetime.iso8601 (#PCDATA)>
<!ELEMENT base64 (#PCDATA)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT superclass (#PCDATA)>
<!ELEMENT item (#PCDATA)>
```

```
<!ELEMENT returnType (#PCDATA)>
<!ELEMENT member (name, value)>
<!ELEMENT struct (member+)>
<!ELEMENT data (value+)>
<!ELEMENT array (data)>
<!ELEMENT value (#PCDATA|i4|int|string|double|datetime.iso8601|base64|
   struct|array)*>
<!ELEMENT attribute (name, value)>
<!ELEMENT timestamp (#PCDATA)>
<!ELEMENT newAddress (#PCDATA)>
<!ELEMENT attributeDescription (name, type, desc*)>
<!ATTLIST attributeDescription
          required (true|false|0|1) "false"
          writable (true|false|0|1) "false"
          allocation (class|instance) "instance">
<!ELEMENT params (param+)>
<!ELEMENT param (name, type, desc*)>
<!ELEMENT methodDescription (name, returnType, params?, desc*)>
<!ATTLIST methodDescription
          allocation (class|instance) "instance">
<!ELEMENT describe (desc*, attributeDescription*, methodDescription*,
  (class*|superclass*), timestamp?)>
<!ELEMENT read (name*|(attribute*, timestamp?))>
<!ELEMENT edit (attribute*|newAddress?)>
<!ELEMENT add (attribute*|newAddress)>
<!ELEMENT search (attribute*|item*)>
<!ELEMENT delete EMPTY>
```

16 Appendix D: Objects in Extended Example

Because JOAP requires some significant examples to define the protocol, an example domain was developed to provide consistency. Readers familiar with UML may find the following diagram useful to illustrate some of the fine points of JOAP listed above.

Listing 30: Object Diagram

```
Train:
      number: i4
      name: string
      location: TrackSegment
      cars: Car[]
      void forward()
      void back()
      void insertCar(Car car, Car before)

      Car:
      trackingNumber: i4
```

```
i4 nextTrackingNumber() {class}

Caboose: Car

Engine: Car
canPull: i4

Boxcar: Car
contents: string

PassengerCar: Car
passengers: i4

Building:
name: string
size: struct (length: i4, width: i4)

TrackSegment:
previous: TrackSegment
next: TrackSegment

Switch:
in: TrackSegment
out: TrackSegment[]
boolean switchTo(TrackSegment)

Station: TrackSegment, Building
```