OBIX Version 1.1

Committee Specification Draft 02

19 December 2013

Specification URIs

This version:

http://docs.oasis-open.org/obix/obix/v1.1/csd02/obix-v1.1-csd02.pdf (Authoritative)
http://docs.oasis-open.org/obix/obix/v1.1/csd02/obix-v1.1-csd02.html
http://docs.oasis-open.org/obix/obix/v1.1/csd02/obix-v1.1-csd02.doc

Previous version:

http://docs.oasis-open.org/obix/obix/v1.1/csprd01/obix-v1.1-csprd01.pdf (Authoritative)
http://docs.oasis-open.org/obix/obix/v1.1/csprd01/obix-v1.1-csprd01.html
http://docs.oasis-open.org/obix/obix/v1.1/csprd01/obix-v1.1-csprd01.doc

Latest version:

http://docs.oasis-open.org/obix/obix/v1.1/obix-v1.1.pdf (Authoritative)
http://docs.oasis-open.org/obix/obix/v1.1/obix-v1.1.html
http://docs.oasis-open.org/obix/obix/v1.1/obix-v1.1.doc

Technical Committee:

OASIS Open Building Information Exchange (oBIX) TC

Chair:

Toby Considine (toby.considine@unc.edu), University of North Carolina at Chapel Hill

Editor:

Craig Gemmill (craig.gemmill@tridium.com), Tridium, Inc.

Additional artifacts:

This prose specification is one component of a Work Product that also includes:

- XML schemas: http://docs.oasis-open.org/obix/obix/v1.1/csd02/schemas/

Related work:

This specification replaces or supersedes:

- *oBIX 1.0*. 5 December 2006. OASIS Committee Specification 01. https://www.oasis-open.org/committees/download.php/21812/obix-1.0-cs-01.pdf.

This specification is related to:

- *Bindings for OBIX: REST Bindings Version 1.0*. Edited by Craig Gemmill and Markus Jung. Latest version. http://docs.oasis-open.org/obix/obix-rest/v1.0/obix-rest-v1.0.html.
- *Bindings for OBIX: SOAP Bindings Version 1.0.* Edited by Markus Jung. Latest version. http://docs.oasis-open.org/obix/obix-soap/v1.0/obix-soap-v1.0.html.
- *Encodings for OBIX: Common Encodings Version 1.0.* Edited by Marcus Jung. Latest version. http://docs.oasis-open.org/obix/obix-encodings/v1.0/obix-encodings-v1.0.html.
- *Bindings for OBIX: Web Socket Bindings Version 1.0*. Edited by Matthias Hub. Latest version. http://docs.oasis-open.org/obix/obix-websocket/v1.0/obix-websocket-v1.0.html.

Abstract:

This document specifies an object model used for machine-to-machine (M2M) communication. Companion documents will specify the protocol bindings and encodings for specific cases.

Status:

This document was last revised or approved by the OASIS Open Building Information Exchange (oBIX) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at http://www.oasis-open.org/committees/obix/.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (http://www.oasis-open.org/committees/obix/ipr.php).

Citation format:

When referencing this specification the following citation format should be used:

[OBIX-v1.1]

OBIX Version 1.1. Edited by Craig Gemmill. 19 December 2013. OASIS Committee Specification Draft 02. http://docs.oasis-open.org/obix/obix/v1.1/csd02/obix-v1.1-csd02.html. Latest version: http://docs.oasis-open.org/obix/obix/v1.1/obix-v1.1.html.

Notices

Table of Contents

Table of Figures

Table of Tables

1 Introduction

OBIX is designed to provide access to the embedded software systems which sense and control the world around us. Historically, integrating to these systems required custom low level protocols, often custom physical network interfaces. The rapid increase in ubiquitous networking and the availability of powerful microprocessors for low cost embedded devices is now weaving these systems into the very fabric of the Internet. Generically the term M2M for Machine-to-Machine describes the transformation occurring in this space because it opens a new chapter in the development of the Web - machines autonomously communicating with each other. The OBIX specification lays the groundwork for building this M2M Web using standard, enterprise-friendly technologies like XML, HTTP, and URIs.

1.1 Terminology

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in **RFC2119**.

1.2 Normative References

PNG W3C Recommendation, “PNG (Portable Network Graphics) Specification”, 1 October 1996. http://www.w3.org/TR/REC-png-multi.html.

RFC2119 Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, March 1997. http://www.ietf.org/rfc/rfc2119.txt.

RFC2246 Dierks, T., Allen, C., “Transport Layer Security (TLS) Protocol Version 1.0”, IETF RFC 2246, January 1999. http://www.ietf.org/rfc/rfc2246.txt.

RFC3986 Berners-Lee, T., Fielding, R., Masinter, L., “Uniform Resource Identifier (URI): Generic Syntax”, IETF RFC 3986, January 2005. http://www.ietf.org/rfc/rfc3986.txt.

SI Units International System of Units (SI), NIST Reference, http://physics.nist.gov/cuu/Units/units.html.

SOA-RM *Reference Model for Service Oriented Architecture 1.0*, October 2006. OASIS Standard. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf.

WS-Calendar *WS-Calendar Version 1.0*, 30 July 2011. OASIS Committee Specification, http://docs.oasis-open.org/ws-calendar/ws-calendar/v1.0/ws-calendar-1.0-spec.html.

WSDL Christensen, E., Curbera, F., Meredith, G., Weerawarana, S., “Web Services Description Language (WSDL), Version 1.1”, W3C Note, 15 March 2001. http://www.w3.org/TR/wsdl.

XLINK DeRose, S., Maler, E., Orchard, D., Walsh, N. “XML Linking Language (XLink) Version 1.1”, May 2010. http://www.w3.org/TR/xlink11/ .

XPOINTER DeRose, S., Maler, E., Daniel Jr., R., “XPointer xpointer() Scheme”, December 2002. http://www.w3.org/TR/xptr-xpointer/.

XML Schema Biron, P.V., Malhotra, A., “XML Schema Part 2: Datatypes Second Edition”, October 2004. http://www.w3.org/TR/xmlschema-2/.

ZoneInfo DB IANA Time Zone Database, 24 September 2013 (latest version), http://www.iana.org/time-zones.

1.3 Non-Normative References

Casing *Capitalization Styles,* Microsoft Developer Network, September, 2013. http://msdn.microsoft.com/en-us/library/x2dbyw72(v=vs.71).aspx.

OBIX REST *Bindings for OBIX: REST Bindings Version 1.0*. Edited by Craig Gemmill and Markus Jung. Latest version. http://docs.oasis-open.org/obix/obix-rest/v1.0/obix-rest-v1.0.html.

OBIX SOAP *Bindings for OBIX: SOAP Bindings Version 1.0.* Edited by Markus Jung. Latest version. http://docs.oasis-open.org/obix/obix-soap/v1.0/obix-soap-v1.0.html.

OBIX Encodings *Encodings for OBIX: Common Encodings Version 1.0.* Edited by Marcus Jung. Latest version. http://docs.oasis-open.org/obix/obix-encodings/v1.0/obix-encodings-v1.0.html.

OBIX WebSockets *Bindings for OBIX: Web Socket Bindings Version 1.0*. Edited by Matthias Hub. Latest version. http://docs.oasis-open.org/obix/obix-websocket/v1.0/obix-websocket-v1.0.html.

RDDL 2.0 Jonathan Borden, Tim Bray, eds. “Resource Directory Description Language (RDDL) 2.0,” January 2004. http://www.openhealth.org/RDDL/20040118/rddl-20040118.html.

REST Fielding, R.T., “Architectural Styles and the Design of Network-based Software Architectures”, Dissertation, University of California at Irvine, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

SOAP Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Nielsen, H., Karmarkar, A., Lafon, Y., “SOAP Version 1.2 (Second Edition)”, W3C Recommendation 27 April 2007. http://www.w3.org/TR/soap12/.

UML *Unified Modeling Language (UML), Version 2.2*, Object Management Group, February, 2009. http://www.omg.org/technology/documents/formal/uml.htm .

XML-ns W3C Recommendation, “Namespaces in XML”, 14 January 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114/.

1.4 Namespace

If an implementation is using the XML Encoding according to the **OBIX Encodings** specification document, the XML namespace URI (see **XML-ns**) that MUST be used is:

```
http://docs.oasis-open.org/obix/ns/201310
```

Dereferencing the above URI will produce the Resource Directory Description Language (**RDDL 2.0**) document that describes this namespace.

1.5 Naming Conventions

Where XML is used, for the names of elements and the names of attributes within XSD files, the names follow the Lower Camel Case convention (see **Casing** for a description of Camel Case), with all names starting with a lower case letter.

1.6 Editing Conventions

For readability, Element names in tables appear as separate words. In the Schema, they follow the rules as described in Section 1.5.

Terms defined in this specification or used from specific cited references are capitalized; the same term not capitalized has its normal English meaning.

All sections explicitly noted as examples are informational and SHALL NOT be considered normative.

All UML and figures are illustrative and SHALL NOT be considered normative.

1.7 Language Conventions

Although several different encodings may be used for representing OBIX data, the most common is XML. Therefore many of the concepts in OBIX are strongly tied to XML concepts. Data objects are represented in XML by XML *documents*. It is important to distinguish the usage of the term *document* in this context from references to this specification document. When “this document” is used, it references this specification document. When “OBIX document” or “XML document” is used, it references an OBIX object, encoded in XML, as per the convention for this (specification) document. When used in the latter context, this could equally be understood to mean an OBIX object encoded in any of the other possible encoding mechanisms.

When expressed in XML, there is a one-to-one-mapping between *Objects* and *elements*. Objects are the fundamental abstraction used by the OBIX data model. Elements are how those Objects are expressed in XML syntax. This specification uses the term *Object* and *sub-Object*, although one can equivalently substitute the term element and sub-element when referencing the XML representation. The term *child* is used to describe an Object that is contained by another Object, and is semantically equivalent to the term *sub-Object*. The two terms are used interchangeably throughout this specification.

1.8 Architectural Considerations

Table 1-1 illustrates the problem space OBIX attempts to address. Each of these concepts is covered in the subsequent sections of the specification as shown.

Concept	Solution	Covered in Sections
Information Model	Representing M2M information in a standard syntax – originally XML but expanded to other technologies	4, 5, 6, 8, 9
Interactions	transferring M2M information over a network	10
Normalization	developing standard representations for common M2M features: points, histories, and alarms	11, 12, 13, 14, 15
Foundation	providing a common kernel for new standards	7, 11

Table 1-1. Problem spaces for OBIX.

1.8.1 Information Model

OBIX defines a common information model to represent diverse M2M systems and an interaction model for their communications. The design philosophy of OBIX is based on a small but extensible data model which maps to a simple fixed syntax. This core model and its syntax are simple enough to capture entirely in one illustration, which is done in Figure 4-1. The object model’s extensibility allows for the definition of new abstractions through a concept called *Contracts*. Contracts are flexible and powerful enough that they are even used to define the majority of the conformance rules in this specification.

1.8.2 Interactions

Once we have a way to represent M2M information in a common format, the next step is to provide standard mechanisms to transfer it over networks for publication and consumption. OBIX breaks networking into two pieces: an abstract request/response model and a series of protocol bindings which implement that model. In Version 1.1 of OBIX, the two goals are accomplished in separate documents: this core specification defines the core model, while several protocol bindings designed to leverage existing Web Service infrastructureare described in companion documents to this specification.

1.8.3 Normalization

There are a few concepts which have broad applicability in systems which sense and control the physical world. Version 1.1 of OBIX provides a normalized representation for three of these, described in Table 1-2.

Concept	Description

Points	Representing a single scalar value and its status – typically these map to sensors, actuators, or configuration variables like a setpoint
Histories	Modeling and querying of time sampled point data. Typically edge devices collect a time stamped history of point values which can be fed into higher level applications for analysis
Alarms	Modeling, routing, and acknowledgment of alarms. Alarms indicate a condition which requires notification of either a user or another application

Table 1-2. Normalization concepts in OBIX.

1.8.4 Foundation

The requirements and vertical problem domains for M2M systems are immensely broad – too broad to cover in one single specification. OBIX is deliberately designed as a fairly low level specification, but with a powerful extension mechanism based on Contracts. The goal of OBIX is to lay the groundwork for a common object model and XML syntax which serves as the foundation for new specifications. It is hoped that a stack of specifications for vertical domains can be built upon OBIX as a common foundation.

1.9 Changes from Version 1.0

Changes to this specification since the initial version 1.0 are listed in Table 1-3 below, along with a brief description.

Add `date`, `time` primitive types and `tz` Facet to the core object model.
Add binary encoding – Note this is now part of the Encodings for OBIX document.
Add support for History Append operation.
Add HTTP content negotiation – Note this is now part of the OBIX REST document.
Add the `of` attribute to the `ref` element type and specify usage of the `is` attribute for `ref`.
Add metadata inclusion for alternate hierarchies (tagging).
Add compact history record encoding.
Add support for alternate history formats.
Add support for concise encoding of long Contract Lists.
Add Delete request semantics.
Clean up references and usage in text, add tables and Table of Tables, capitalization of important words.
Add conformance clauses.
Move Lobby earlier in document and add Bindings, Encodings, and Models sections.

Table 1-3. Changes from Version 1.0.

2 Quick Start [non-normative]

This chapter is for those eager to jump right into OBIX in all its angle bracket glory. The best way to begin is to take a simple example that anybody is familiar with – the staid thermostat. Let's assume we have a very simple thermostat. It has a temperature sensor which reports the current space temperature and it has a setpoint that stores the desired temperature. Let's assume our thermostat only supports a heating mode, so it has a variable that reports if the furnace should currently be on. Let's take a look at what our thermostat might look like in OBIX XML:

```
<obj href="http://myhome/thermostat">
  <real name="spaceTemp" unit="obix:units/fahrenheit" val="67.2"/>
  <real name="setpoint" unit="obix:units/fahrenheit" val="72.0"/>
  <bool name="furnaceOn" val="true"/>
</obj>
```

The first thing to notice is the **Information Model**: there are three element types – `obj`, `real`, and `bool`. The root `obj` element models the entire thermostat. Its `href` attribute identifies the URI for this OBIX document. The thermostat Object has three child Objects, one for each of the thermostat's variables. The `real` Objects store our two floating point values: space temperature and setpoint. The `bool` Object stores a boolean variable for furnace state. Each sub-element contains a `name` attribute which defines the role within the parent. Each sub-element also contains a `val` attribute for the current value. Lastly we see that we have annotated the temperatures with an attribute called `unit` so we know they are in Fahrenheit, not Celsius (which would be one hot room). The OBIX specification defines several of these annotations which are called *Facets*.

How did we obtain this Object? The OBIX specification leverages commonly available networking technologies and concepts for defining **Interactions** between devices. The thermostat implements an OBIX Server, and we can use an OBIX Client to issue a request for the thermostat's data, by specifying its *uri*. This concept is well understood in the world of M2M so OBIX requires no new knowledge to implement.

In real life, we wish to represent **Normalized** information from devices. In most cases sensor and actuator variables (called *Points*) imply more semantics than a simple scalar value. In the example of our thermostat, in addition to the current space temperature, it also reports the setpoint for desired temperature and whether it is trying to command the furnace on. In other cases such as alarms, it is desirable to standardize a complex data structure. OBIX captures these concepts into *Contracts*. Contracts allow us to tag Objects with normalized semantics and structure.

Let's suppose our thermostat's sensor is reading a value of -412°F? Clearly our thermostat is busted, so it should report a fault condition. Let's rewrite the XML to include the status Facet and to provide additional semantics using Contracts:

```
<obj href="http://myhome/thermostat/">

  <!-- spaceTemp point -->
  <real name="spaceTemp" is="obix:Point"
        val="-412.0" status="fault"
        unit="obix:units/fahrenheit"/>

  <!-- setpoint point -->
  <real name="setpoint" is="obix:Point"
        val="72.0"
        unit="obix:units/fahrenheit"/>

  <!-- furnaceOn point -->
  <bool name="furnaceOn" is="obix:Point" val="true"/>

</obj>
```

Notice that each of our three scalar values are tagged as `obix:Points` via the `is` attribute. This is a standard Contract defined by OBIX for representing normalized point information. By implementing these Contracts, clients immediately know to semantically treat these objects as points.

Contracts play a pivotal role in OBIX because they provide a **Foundation** for building new abstractions upon the core object model. Contracts are just normal objects defined using standard OBIX. In fact, the following sections defining the core OBIX object model are expressed using Contracts. One can see how easily this approach allows for definition of the key parts of this model, or any model that builds upon this model.

3 Architecture

The OBIX architecture is based on the design philosophies and principles in Table 3-1.

Philosophy	Usage/Description
Object Model	A concise object model used to define all OBIX information
Encodings	Sets of rules for representing the object model in certain common formats
URIs	Uniform Resource Identifiers are used to identify information within the object model
REST	A small set of verbs is used to access objects via their URIs and transfer their state
Contracts	A template model for expressing new OBIX “types”
Extensibility	Providing for consistent extensibility using only these concepts

Table 3-1. Design philosophies and principles for OBIX.

3.1 Object Model

All information in OBIX is represented using a small, fixed set of primitives. The base abstraction for these primitives is called *Object*. An Object can be assigned a URI and all Objects can contain other Objects.

3.2 Encodings

A necessary feature of OBIX is a set of simple syntax rules to represent the underlying object model. XML is a widely used language with well-defined and well-understood syntax that maps nicely to the OBIX object model. The rest of this specification will use XML as the example encoding, because it is easily human-readable, and serves to clearly demonstrate the concepts presented. The syntax used is normative. Implementations using an XML encoding MUST conform to this syntax and representation of elements.

When encoding OBIX objects in XML, each of the object types map to one type of element. The Value Objects represent their data value using the `val` attribute (see Section 4.2.1 for a full description of Value Objects). All other aggregation is simply nesting of elements. A simple example to illustrate this concept is the Brady family from the TV show *The Brady Bunch*:

```
<obj href="http://bradybunch/people/Mike-Brady/">
  <obj name="fullName">
    <str name="first" val="Mike"/>
    <str name="last" val="Brady"/>
  </obj>
  <int name ="age" val="45"/>
  <ref name="spouse" href="/people/Carol-Brady"/>
  <list name="children">
    <ref href="/people/Greg-Brady"/>
    <ref href="/people/Peter-Brady"/>
    <ref href="/people/Bobby-Brady"/>
    <ref href="/people/Marsha-Brady"/>
    <ref href="/people/Jan-Brady"/>
    <ref href="/people/Cindy-Brady"/>
  </list>
</obj>
```

Note in this simple example how the `href` attribute specifies URI references which may be used to fetch more information about the object. Names and hrefs are discussed in detail in Section 6.

3.3 URIs

No architecture is complete without some sort of naming system. In OBIX everything is an object, so we need a way to name objects. Since OBIX is really about making information available over the web using XML, it makes sense to leverage the URI (Uniform Resource Identifier) as defined in **RFC3986**. URIs are the standard way to identify "resources" on the web.

Since OBIX is used to interact with control systems over the web, we use the URL to identify each resource. Just as we assume an XML encoding and a REST binding for all examples in this document, so too we assume a URL using the Hypertext Transfer Protocol (URLs beginning with http:) beginning with HTTP. This is not meant to forbid the use of secure transfer (https:) or of other protocols (ws:). Neither are the examples are meant to forbid the use of alternate ports. The URLs in examples in this specification are for illustration only. Often URIs also provide information about how to fetch their resource - that's why they are often called URLs (Uniform Resource Locator). From a practical perspective if a vendor uses HTTP URIs to identify their objects, you can most likely just do a simple HTTP GET to fetch the OBIX document for that object. But technically, fetching the contents of a URI is a protocol binding issue discussed in later chapters.

The value of URIs are that they have numerous defined and commonly understood rules for manipulating them. For example URIs define which characters are legal and which are illegal. Of great value to OBIX is *URI references* which define a standard way to express and normalize relative URIs. In addition, most programming environments have libraries to manage URIs so developers don't have to worry about managing the details of normalization.

3.4 REST

Objects identified with URIs and passed around as XML documents may sound a lot like REST – and this is intentional. REST stands for REpresentational State Transfer and is an architectural style for web services that mimics how the World Wide Web works. The WWW is basically a big web of HTML documents all hyperlinked together using URIs. Likewise, OBIX is basically a big web of XML object documents hyperlinked together using URIs. Because REST is such a key concept in OBIX, it is not surprising that a REST binding is a core part of the specification. The specification of this binding is defined in the **OBIX REST** document.

REST is really more of a design style, than a specification. REST is resource centric as opposed to method centric - resources being OBIX objects. The methods actually used tend to be a very small fixed set of verbs used to work generically with all resources. In OBIX all network requests boil down to four request types:

- **Read**: an object
- **Write**: an object
- **Invoke**: an operation
- **Delete**: an object

3.5 Contracts

In every software domain, patterns start to emerge where many different object instances share common characteristics. For example in most systems that model people, each person probably has a name, address, and phone number. In vertical domains we may attach domain specific information to each person. For example an access control system might associate a badge number with each person.

In object oriented systems we capture these patterns into classes. In relational databases we map them into tables with typed columns. In OBIX these patterns are modeled using a concept called *Contracts*, which are standard OBIX objects used as a template. Contracts provide greater flexibility than a strongly typed schema language, without the overhead of introducing new syntax. A Contract document is parsed just like any other OBIX document. In formal terms, Contracts are a combination of prototype based inheritance and mixins.

Why do we care about trying to capture these patterns? The most important use of Contracts is by the OBIX specification itself to define new standard abstractions. It is just as important for everyone to agree on normalized semantics as it is on syntax. Contracts also provide the definitions needed to map to classes in an object-oriented system, or tables in a relational database.

3.6 Extensibility

We want to use OBIX as a foundation for developing new abstractions in vertical domains. We also want to provide extensibility for vendors who implement OBIX across legacy systems and new product lines. Additionally, it is common for a device to ship as a blank slate and be completely programmed in the field. This leaves us with a mix of standards based, vendor based, and even project based extensions.

The principle behind OBIX extensibility is that anything new is defined strictly in terms of Objects, URIs, and Contracts. To put it another way - new abstractions do not introduce any new XML syntax or functionality that client code is forced to care about. New abstractions are always modeled as standard trees of OBIX objects, just with different semantics. That does not mean that higher level application code never changes to deal with new abstractions. But the core stack that deals with networking and parsing should not have to change to accommodate a new type.

This extensibility model is similar to most mainstream programming languages such as Java or C#. The syntax of the core language is fixed with a built in mechanism to define new abstractions. Extensibility is achieved by defining new class libraries using the language's fixed syntax. This means the compiler need not be updated every time someone adds a new class.

4 Object Model

The OBIX specification is based on a small, fixed set of object types. The OBIX object model is summarized in Figure 4-1. It consists of a common base Object (`obix:obj`) type, and includes 16 derived types. Section 4.1 describes the associated properties called *Facets* that each type may have. Section 4.2 describes each of the core OBIX types, including the rules for their usage and interpretation. Additional rules defining complex behaviors such as naming and Contract inheritance are described in Sections 6 and 7. These sections are essential to a full understanding of the object model.

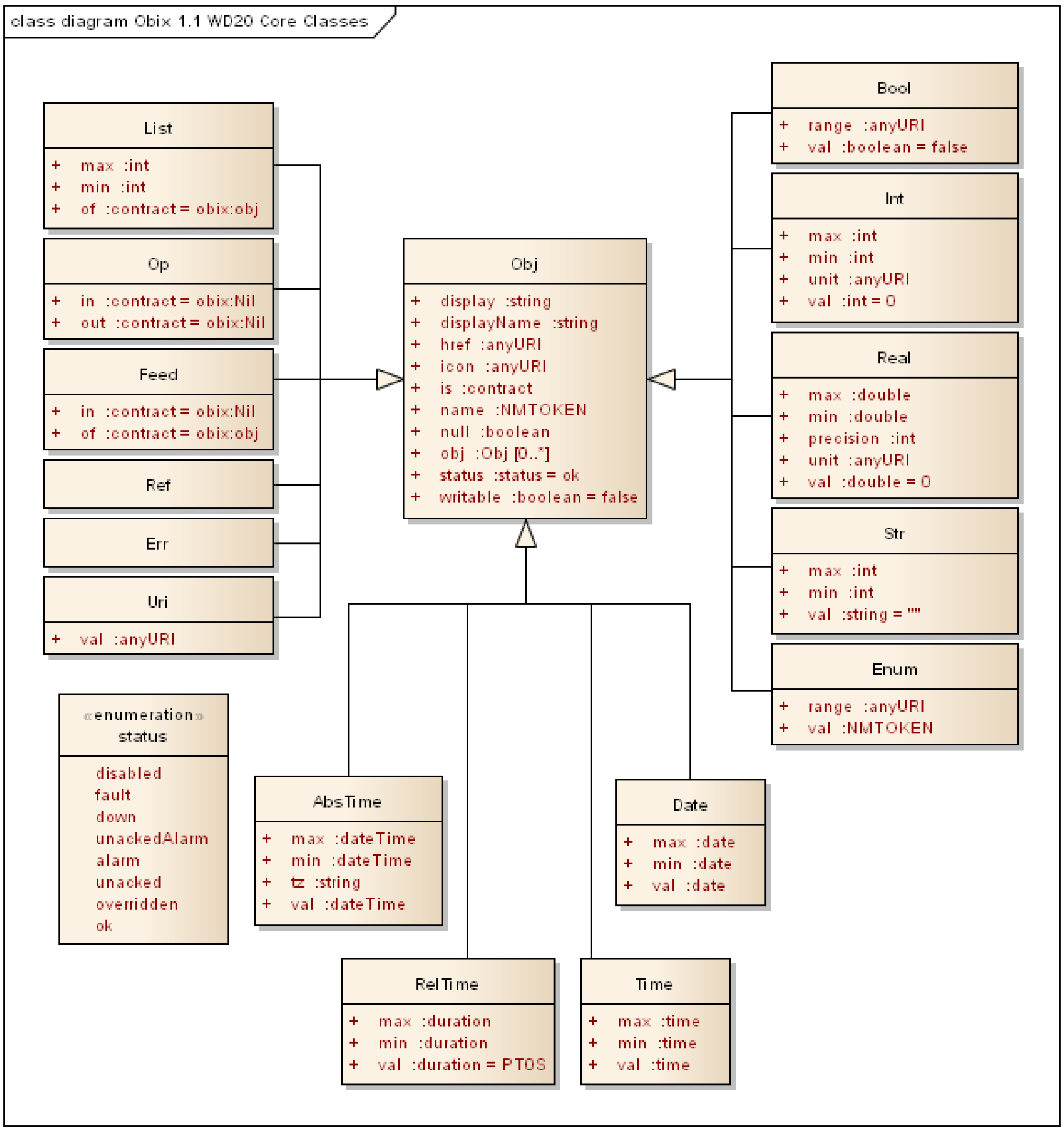

Figure 4-1 The OBIX primitive object hierarchy.

4.1 obj

The root abstraction in OBIX is *Object*. Every type in OBIX is a derivative of Object. Any Object or its derivatives can contain other Objects. The properties supported on Object, and therefore on any derivative type, are listed in Table 4-1.

Property	Description
name	Defines the Object's purpose in its parent Object (discussed in Section 6). Names of Objects SHOULD be in Camel case per **Casing**.
href	Provides a URI reference for identifying the Object (discussed in Section 6).
is	Defines the Contracts the Object implements (discussed in Section 7).
null	Supports the concept of null Objects (discussed in Section 4.1.1 and in Section 7.4).
val	Stores the actual value of the object, used only with value-type Objects (`bool`, `int`, `real`, `str`, `enum`, `abstime`, `reltime`, `date`, `time`, and `uri`). The literal representation of values maps to **XML Schema**, indicated in the following sections via the "`xs:`" prefix.
Facets	A set of properties used to provide meta-data about the Object (discussed in Section 4.1.2).

Table 4-1. Base properties of OBIX Object type.

As stated in Section 3.2, the expression of Objects in an XML encoding is through XML elements. The OBIX Object type is expressed through the `obj` element. The properties of an Object are expressed through XML attributes of the element. The full set of rules for encoding OBIX in XML is contained in the **OBIX Encodings** document. The term `obj` as used in this specification represents an OBIX Object in general, regardless of how it is encoded.

The Contract definition of Object, as expressed by an `obj` element is:

```
<obj href="obix:obj" null="false" writable="false" status="ok" />
```

4.1.1 Null

All Objects support the concept of *null*. Null is the absence of a value, meaning that this Object has no value, has not been configured or initialized, or is otherwise not defined. Null is indicated using the `null` attribute with a boolean value. All Objects default null to false with the exception of `enum`, `abstime`, `date`, and `time` (since any other default would be confusing). An example of a null `abstime` Object is:

```
<abstime name=”startTime” displayName=”Start Time”/>
```

Null is inherited from Contracts a little differently than other attributes. See Section 7.4.3 for details.

4.1.2 Facets

All Objects can be annotated with a predefined set of attributes called *Facets*. Facets provide additional meta-data about the Object. The set of available Facets is: `displayName`, `display`, `icon`, `min`, `max`, `precision`, `range`, `status`, `tz`, `unit`, `writable`, `of`, `in`, and `out`. Although OBIX predefines a number of Facets, vendors MAY add additional Facets. Vendors that wish to annotate Objects with additional Facets SHOULD use XML namespace qualified attributes.

4.1.3 displayName

The `displayName` Facet provides a localized human readable name of the Object stored as an `xs:string`:

```
<obj name="spaceTemp" displayName="Space Temperature"/>
```

Typically the `displayName` Facet SHOULD be a localized form of the `name` attribute. There are no restrictions on `displayName` overrides from the Contract (although it SHOULD be uncommon since `displayName` is just a human friendly version of `name`).

4.1.4 display

The `display` Facet provides a localized human readable description of the Object stored as an `xs:string`:

```
<bool name="occupied" val="false" display="Unoccupied"/>
```

There are no restrictions on `display` overrides from the Contract.

The `display` attribute serves the same purpose as Object.toString() in Java or C#. It provides a general way to specify a string representation for all Objects. In the case of value Objects (like `bool` or `int`) it SHOULD provide a localized, formatted representation of the `val` attribute.

4.1.5 icon

The `icon` Facet provides a URI reference to a graphical icon which may be used to represent the Object in an user agent:

```
<obj icon="/icons/equipment.png"/>
```

The contents of the `icon` attribute MUST be a URI to an image file. The image file SHOULD be a 16x16 PNG file, defined in the **PNG** specification. There are no restrictions on `icon` overrides from the Contract.

4.1.6 min

The `min` Facet is used to define an inclusive minimum value:

```
<int min="5" val="6"/>
```

The contents of the `min` attribute MUST match its associated `val` type. The `min` Facet is used with `int`, `real` , `abstime`, `date`, `time`, and `reltime` to define an inclusive lower limit of the value space. It is used with `str` to indicate the minimum number of Unicode characters of the string. It is used with `list` to indicate the minimum number of child Objects (named or unnamed). Overrides of the `min` Facet may only narrow the value space using a larger value. The `min` Facet MUST never be greater than the `max` Facet (although they MAY be equal).

4.1.7 max

The `max` Facet is used to define an inclusive maximum value:

```
<real max="70" val="65"/>
```

The contents of the `max` attribute MUST match its associated `val` type. The `max` Facet is used with `int`, `real`, `abstime`, `date`, `time`, and `reltime` to define an inclusive upper limit of the value space. It is used with `str` to indicate the maximum number of Unicode characters of the string. It is used with `list` to indicate the maximum number of child Objects (named or unnamed). Overrides of the `max` Facet may only narrow the value space using a smaller value. The `max` Facet MUST never be less than the `min` Facet (although they MAY be equal).

4.1.8 precision

The `precision` Facet is used to describe the number of decimal places to use for a `real` value:

```
<real precision="2" val="75.04"/>
```

The contents of the `precision` attribute MUST be `xs:int`. The value of the `precision` attribute equates to the number of meaningful decimal places. In the example above, the value of 2 indicates two meaningful decimal places: “75.04”. Typically precision is used by client applications which do their own formatting of `real` values. There are no restrictions on `precision` overrides.

4.1.9 range

The `range` Facet is used to define the value space of an enumeration. A `range` attribute is a URI reference to an `obix:Range` Object (see section 11.2 for the definition). It is used with the `bool` and `enum` types:

```
<enum range="/enums/OffSlowFast" val="slow"/>
```

The override rule for `range` is that the specified range MUST inherit from the Contract’s range. Enumerations are unusual in that specialization of an enum usually involves adding new items to the range. Technically this is widening the enum’s value space, rather than narrowing it. But in practice, adding items into the range is what we desire.

4.1.10 status

The `status` Facet is used to annotate an Object about the quality and state of the information:

```
<real val="67.2" status="alarm"/>
```

Status is an enumerated string value with one of the following values from Table 4-2 (ordered by priority):

Status	Description
`disabled`	This state indicates that the Object has been disabled from normal operation (out of service). In the case of operations and feeds, this state is used to disable support for the operation or feed.
`fault`	The `fault` state indicates that the data is invalid or unavailable due to a failure condition - data which is out of date, configuration problems, software failures, or hardware failures. Failures involving communications should use the `down` state.
`down`	The `down` state indicates a communication failure.
`unackedAlarm`	The `unackedAlarm` state indicates there is an existing alarm condition which has not been acknowledged by a user – it is the combination of the `alarm` and `unacked` states. The difference between `alarm` and `unackedAlarm` is that `alarm` implies that a user has already acknowledged the alarm or that no human acknowledgement is necessary for the alarm condition. The difference between `unackedAlarm` and `unacked` is that the Object has returned to a normal state.
`alarm`	This state indicates the Object is currently in the alarm state. The alarm state typically means that an Object is operating outside of its normal boundaries. In the case of an analog point this might mean that the current value is either above or below its configured limits. Or it might mean that a digital sensor has transitioned to an undesired state. See Alarming (Section 15) for additional information.
`unacked`	The `unacked` state is used to indicate a past alarm condition which remains unacknowledged.
`overridden`	The `overridden` state means the data is ok, but that a local override is currently in effect. An example of an override might be the temporary override of a setpoint from its normal scheduled setpoint.
`ok`	The `ok` state indicates normal status. This is the assumed default state for all Objects.

Table 4-2. Status enumerations in OBIX.

Status MUST be one of the enumerated strings above. It might be possible in the native system to exhibit multiple status states simultaneously, however when mapping to OBIX the highest priority status SHOULD be chosen – priorities are ranked from top (disabled) to bottom (ok).

4.1.11 tz

The `tz` Facet is used to annotate an `abstime`, `date`, or `time` Object with a timezone. The value of a `tz` attribute is a *zoneinfo* string identifier, as specified in the IANA Time Zone (**ZoneInfo DB**) database. The zoneinfo database defines the current and historical rules for each zone including its offset from UTC and the rules for calculating daylight saving time. OBIX does not define a Contract for modeling timezones, instead it just references the zoneinfo database using standard identifiers. It is up to OBIX enabled software to map zoneinfo identifiers to the UTC offset and daylight saving time rules.

The following rules are used to compute the timezone of an `abstime`, `date`, or `time` Object:

1. If the `tz` attribute is specified, set the timezone to `tz`;
2. Otherwise, if the Contract defines an inherited `tz` attribute, set the timezone to the inherited `tz` attribute;
3. Otherwise, set the timezone to the server's timezone as defined by the lobby's `About.tz`.

When using timezones, an implementation MUST specify the timezone offset within the value representation of an `abstime` or `time` Object. It is an error condition for the `tz` Facet to conflict with the timezone offset. For example, New York has a -5 hour offset from UTC during standard time and a -4 hour offset during daylight saving time:

```
<abstime val="2007-12-25T12:00:00-05:00" tz="America/New_York"/>
<abstime val="2007-07-04T12:00:00-04:00" tz="America/New_York"/>
```

4.1.12 unit

The `unit` Facet defines a unit of measurement in the **SI Units** system. A unit attribute is a URI reference to an `obix:Unit` Object (see section 11.5 for the Contract definition). It is used with the `int` and `real` types:

```
<real unit="obix:units/fahrenheit" val="67.2"/>
```

It is recommended that the `unit` Facet not be overridden if declared in a Contract. If it is overridden, then the override SHOULD use a `Unit` Object with the same dimensions as the Contract (it must measure the same physical quantity).

4.1.13 writable

The `writable` Facet specifies if this Object can be written by the client. If false (the default), then the Object is read-only. It is used with all types except `op` and `feed`:

```
<str name="userName" val="jsmith"     writable="false"/>
<str name="fullName" val="John Smith" writable="true"/>
```

The `writable` Facet describes only the ability of clients to modify this Object's value, not the ability of clients to add or remove children of this Object. Servers MAY allow addition or removal of child Objects independently of the writability of existing objects. If a server does not support addition or removal of Object children through writes, it MUST return an appropriate error response (see Section 10.2 for details).

4.1.14 of

The `of` Facet specifies the type of child Objects contained by this Object. This Facet is used with `list` and `ref` types. The use of this Facet for each case is explained with the definition of the type, in Section 4.2.2 for `list` and 4.2.3 for `ref`.

4.1.15 in

The `in` Facet specifies the input argument type used by this Object. This Facet is used with `op` and `feed` types. Its use is described with the definition of those types in Section 4.2.5 for `op` and 4.2.6 for `feed`.

4.1.16 out

The `out` Facet specifies the output argument type used by this Object. This Facet is used with the `op` type. Its use is described with the definition of that type in Section 4.2.5.

4.2 Core Types

OBIX defines a handful of core types which derive from Object. Certain types are allowed to have a `val` attribute and are called “value” types. This concept is expressed in object-oriented terms by using an “abstract” `val` type, and the value subtypes inheriting the `val` behavior from their supertype.

4.2.1 val

A special type of Object called a *Value* Object is used to store a piece of simple information. The `val` type is not directly used (it is “abstract”). It simply reflects that the type may contain a `val` attribute, as it is used to represent an object that has a specific value. The different Value Object types defined for OBIX are listed in Table 4-3.

Type Name	Usage
bool	stores a boolean value – true or false
int	stores an integer value
real	stores a floating point value
str	stores a UNICODE string
enum	stores an enumerated value within a fixed range
abstime	stores an absolute time value (timestamp)
reltime	stores a relative time value (duration or time span)
date	stores a specific date as day, month, and year
time	stores a time of day as hour, minutes, and seconds
uri	stores a Universal Resource Identifier

Table 4-3. Value Object types.

Note that any Value Object can also contain sub-Objects.

4.2.1.1 bool

The `bool` type represents a boolean condition of either true or false. Its `val` attribute maps to `xs:boolean` defaulting to false. The literal value of a `bool` MUST be “true” or “false” (the literals “1” and “0” are not allowed). The Contract definition is:

```
<bool href="obix:bool" is="obix:obj" val="false" null="false"/>
```

An example:

```
<bool val="true"/>
```

4.2.1.2 int

The `int` type represents an integer number. Its `val` attribute maps to `xs:long` as a 64-bit integer with a default of 0. The Contract definition is:

```
<int href="obix:int" is="obix:obj" val="0" null="false"/>
```

An example:

```
<int val="52"/>
```


4.2.1.3 real

The `real` type represents a floating point number. Its `val` attribute maps to `xs:double` as a IEEE 64-bit floating point number with a default of 0. The Contract definition is:

```
<real href="obix:real" is="obix:obj" val="0" null="false"/>
```

An example:

```
<real val="41.06"/>
```

4.2.1.4 str

The `str` type represents a string of Unicode characters. Its `val` attribute maps to `xs:string` with a default of the empty string. The Contract definition is:

```
<str href="obix:str" is="obix:obj" val="" null="false"/>
```

An example:

```
<str val="hello world"/>
```

4.2.1.5 enum

The `enum` type is used to represent a value which must match a finite set of values. The finite value set is called the *range*. The `val` attribute of an `enum` is represented as a string key using `xs:string`. Enums default to null. The range of an `enum` is declared via Facets using the `range` attribute. The Contract definition is:

```
<enum href="obix:enum" is="obix:obj" val="" null="true"/>
```

An example:

```
<enum range="/enums/OffSlowFast" val="slow"/>
```

In this example, the `val` attribute is specified, so the `null` attribute is implied to be false. See Section 7.4.3 for details on the inheritance of the `null` attribute.

4.2.1.6 abstime

The `abstime` type is used to represent an absolute point in time. Its `val` attribute maps to `xs:dateTime`, with the exception that it MUST contain the timezone. According to XML Schema Part 2 section 3.2.7.1, the lexical space for abstime is:

```
'-'? yyyy '-' mm '-' dd 'T' hh ':' mm ':' ss ('.' s+)? (zzzzzz)
```

Abstimes default to null. The Contract definition is:

```
<abstime href="obix:abstime" is="obix:obj" val="1970-01-01T00:00:00Z" null="true"/>
```

An example for 9 March 2005 at 1:30PM GMT:

```
<abstime val="2005-03-09T13:30:00Z"/>
```

In this example, the `val` attribute is specified, so the `null` attribute is implied to be false. See Section 7.4.3 for details on the inheritance of the `null` attribute.

The timezone offset is required, so the abstime can be used to uniquely relate the abstime to UTC. The optional `tz` Facet is used to specify the timezone as a zoneinfo identifier. This provides additional context about the timezone, if available. The timezone offset of the `val` attribute MUST match the offset for the timezone specified by the `tz` Facet, if it is also used. See the `tz` Facet section for more information.

4.2.1.7 reltime

The `reltime` type is used to represent a relative duration of time. Its `val` attribute maps to `xs:duration` with a default of 0 seconds. The Contract definition is:

```
<reltime href="obix:reltime" is="obix:obj" val="PT0S" null="false"/>
```

An example of 15 seconds:

```
<reltime val="PT15S"/>
```


4.2.1.8 date

The `date` type is used to represent a day in time as a day, month, and year. Its `val` attribute maps to `xs:date`. According to XML Schema Part 2 section 3.2.9.1, the lexical space for `date` is:

```
'-'? yyyy '-' mm '-' dd
```

Date values in OBIX MUST omit the timezone offset and MUST NOT use the trailing "Z". Only the tz attribute SHOULD be used to associate the date with a timezone. Date Objects default to null. The Contract definition is:

```
<date href="obix:date" is="obix:obj" val="1970-01-01" null="true"/>
```

An example for 26 November 2007:

```
<date val="2007-11-26"/>
```

In this example, the `val` attribute is specified, so the `null` attribute is implied to be false. See Section 7.4.3 for details on the inheritance of the `null` attribute.

The `tz` Facet is used to specify the timezone as a zoneinfo identifier. See the `tz` Facet section for more information.

4.2.1.9 time

The `time` type is used to represent a time of day in hours, minutes, and seconds. Its `val` attribute maps to `xs:time`. According to XML Schema Part 2 section 3.2.8, the lexical space for `time` is the left truncated representation of `xs:dateTime`:

```
hh ':' mm ':' ss ('.' s+)?
```

Time values in OBIX MUST omit the timezone offset and MUST NOT use the trailing "Z". Only the tz attribute SHOULD be used to associate the time with a timezone. Time Objects default to null. The Contract definition is:

```
<time href="obix:time" is="obix:obj" val="00:00:00" null="true"/>
```

An example for 4:15 AM:

```
<time val="04:15:00"/>
```

In this example, the `val` attribute is specified, so the `null` attribute is implied to be false. See Section 7.4.3 for details on the inheritance of the `null` attribute.

The `tz` Facet is used to specify the timezone as a zoneinfo identifier. See the `tz` Facet section for more information.

4.2.1.10 uri

The `uri` type is used to store a URI reference. Unlike a plain old `str`, a `uri` has a restricted lexical space as defined by **RFC3986** and the XML Schema `xs:anyURI` type. OBIX servers MUST use the URI syntax described by **RFC3986** for identifying resources. OBIX clients MUST be able to navigate this URI syntax. Most URIs will also be a URL, meaning that they identify a resource and how to retrieve it (typically via HTTP). The Contract definition is:

```
<uri href="obix:uri" is="obix:obj" val="" null="false"/>
```

An example for the OBIX home page:

```
<uri val="http://obix.org/" />
```

4.2.2 list

The `list` type is a specialized Object type for storing a list of other Objects. The primary advantage of using a `list` versus a generic `obj` is that `lists` can specify a common Contract for their contents using the `of` attribute. If specified, the `of` attribute MUST be a list of URIs formatted as a Contract List. The definition of `list` is:

```
<list href="obix:list" is="obix:obj" of="obix:obj"/>
```

An example list of strings:


```
<list of="obix:str">
  <str val="one"/>
  <str val="two"/>
</list>
```

Because lists typically have constraints on the URIs used for their child elements, they use special semantics for adding children. Lists are discussed in greater detail along with Contracts in section 7.8.

4.2.3 ref

The ref type is used to create an external reference to another OBIX Object. It is the OBIX equivalent of the HTML anchor tag. The Contract definition is:

```
<ref href="obix:ref " is="obix:obj"/>
```

A ref element MUST always specify an href attribute. A ref element SHOULD specify the type of the referenced object using the is attribute. A ref element referencing a list (is="obix:list") SHOULD specify the type of the Objects contained in the list using the of attribute. References are discussed in detail in section 9.2.

4.2.4 err

The err type is a special Object used to indicate an error. Its actual semantics are context dependent. Typically err Objects SHOULD include a human readable description of the problem via the display attribute. The Contract definition is:

```
<err href="obix:err" is="obix:obj"/>
```

4.2.5 op

The op type is used to define an operation. All operations take one input Object as a parameter, and return one Object as an output. The input and output Contracts are defined via the in and out attributes. The Contract definition is:

```
<op href="obix:op" is="obix:obj" in="obix:Nil" out="obix:Nil"/>
```

Operations are discussed in detail in Section 8.

4.2.6 feed

The feed type is used to define a topic for a feed of events. Feeds are used with Watches to subscribe to a stream of events such as alarms. A feed SHOULD specify the event type it fires via the of attribute. The in attribute can be used to pass an input argument when subscribing to the feed (a filter for example).

```
<feed href="obix:feed" is="obix:obj" in="obix:Nil" of="obix:obj"/>
```

Feeds are subscribed via Watches. This is discussed in Section 12.

5 Lobby

All OBIX servers MUST provide an Object which implements `obix:Lobby`. The `Lobby` Object serves as the central entry point into an OBIX server, and lists the URIs for other well-known Objects defined by the OBIX Specification. Theoretically all a client needs to know to bootstrap discovery is one URI for the `Lobby` instance. By convention this URI is "http://<server-ip-address>/obix", although vendors are certainly free to pick another URI. The Lobby Contract is:

```
<obj href="obix:Lobby">
  <ref name="about" is="obix:About"/>
  <op  name="batch" in="obix:BatchIn" out="obix:BatchOut"/>
  <ref name="watchService" is="obix:WatchService"/>
  <list name="models" of="obix:uri" null="true"/>
  <list name="encodings" of="obix:str" null="true"/>
  <list name="bindings" of="obix:str" null="true"/>
</obj>
```

The `Lobby` instance is where implementers SHOULD place vendor-specific Objects used for data and service discovery. The standard Objects defined in the Lobby Contract are described in the following Sections.

5.1 About

The `obix:About` Object is a standardized list of summary information about an OBIX server. Clients can discover the `About` URI directly from the `Lobby`. The `About` Contract is:

```
<obj href="obix:About">

  <str name="obixVersion"/>

  <str name="serverName"/>
  <abstime name="serverTime"/>
  <abstime name="serverBootTime"/>

  <str name="vendorName"/>
  <uri name="vendorUrl"/>

  <str name="productName"/>
  <str name="productVersion"/>
  <uri name="productUrl"/>

  <str name="tz"/>
</obj>
```

The following children provide information about the OBIX implementation:

- **`obixVersion`**: specifies which version of the OBIX specification the server implements. This string MUST be a list of decimal numbers separated by the dot character (Unicode 0x2E). The current version string is "1.1".

The following children provide information about the server itself:

- **`serverName`**: provides a short localized name for the server.
- **`serverTime`**: provides the server's current local time.
- **`serverBootTime`**: provides the server's start time - this SHOULD be the start time of the OBIX server software, not the machine's boot time.

The following children provide information about the server's software vendor:

- **`vendorName`**: the company name of the vendor who implemented the OBIX server software.
- **`vendorUrl`**: a URL to the vendor's website.

The following children provide information about the software product running the server:

- **productName**: with the product name of OBIX server software.
- **productUrl**: a URL to the product's website.
- **productVersion**: a string with the product's version number. Convention is to use decimal digits separated by dots.

The following children provide additional miscellaneous information:

- **tz**: specifies a zoneinfo identifier for the server's default timezone.

5.2 Batch

The Lobby defines a batch operation which is used to batch multiple network requests together into a single operation. Batching multiple requests together can often provide significant performance improvements over individual round-robin network requests. As a general rule, one big request will always out-perform many small requests over a network.

A batch request is an aggregation of read, write, and invoke requests implemented as a standard OBIX operation. At the protocol binding layer, it is represented as a single invoke request using the Lobby.batch URI. Batching a set of requests to a server MUST be processed semantically equivalent to invoking each of the requests individually in a linear sequence.

The batch operation inputs a BatchIn Object and outputs a BatchOut Object:

```
<list href="obix:BatchIn" of="obix:uri"/>

<list href="obix:BatchOut" of="obix:obj"/>
```

The BatchIn Contract specifies a list of requests to process identified using the Read, Write, or Invoke Contract:

```
<uri href="obix:Read"/>

<uri href="obix:Write">
  <obj name="in"/>
</uri>

<uri href="obix:Invoke">
  <obj name="in"/>
</uri>
```

The BatchOut Contract specifies an ordered list of the response Objects to each respective request. For example the first Object in BatchOut must be the result of the first request in BatchIn. Failures are represented using the err Object. Every uri passed via BatchIn for a read or write request MUST have a corresponding result obj in BatchOut with an href attribute using an identical string representation from BatchIn (no normalization or case conversion is allowed).

It is up to vendors to decide how to deal with partial failures. In general idempotent requests SHOULD indicate a partial failure using err, and continue processing additional requests in the batch. If a server decides not to process additional requests when an error is encountered, then it is still REQUIRED to return an err for each respective request not processed.

Let's look at a simple example:

```
<list is="obix:BatchIn">
  <uri is="obix:Read" val="/someStr"/>
  <uri is="obix:Read" val="/invalidUri"/>
  <uri is="obix:Write" val="/someStr">
    <str name="in" val="new string value"/>
  </uri>
</list>

<list is="obix:BatchOut">
  <str href="/someStr" val="old string value"/>
  <err href="/invalidUri" is="obix:BadUriErr" display="href not found"/>
  <str href="/someStr" val="new string value">
</list>
```


In this example, the batch request is specifying a read request for “/someStr” and “/invalidUri”, followed by a write request to “/someStr”. Note that the write request includes the value to write as a child named “in”. The server responds to the batch request by specifying exactly one Object for each request URI. The first read request returns a `str` Object indicating the current value identified by “/someStr”. The second read request contains an invalid URI, so the server returns an `err` Object indicating a partial failure and continues to process subsequent requests. The third request is a write to “someStr”. The server updates the value at “someStr”, and returns the new value. Note that because the requests are processed in order, the first request provides the original value of “someStr” and the third request contains the new value. This is exactly what we would expect had we processed each of these requests individually.

5.3 WatchService

The WatchService is an important mechanism for providing data from a Server. As such, this specification devotes an entire Section to the description of Watches, and of the WatchService. Section 12 covers Watches in detail.

5.4 Server Metadata

Several components of the Lobby provide additional information about the server’s implementation of the OBIX specification. This is to be used by clients to allow them to tailor their interaction with the server based on mutually interoperable capabilities. The following subsections describe these components.

5.4.1 Models

Any semantic models, such as tag dictionaries, used by the Server for presenting metadata about its Objects MUST be identified in the Lobby in the `models` element, which is a `list` of `uris`. The name of each `uri` MUST be the name that is referenced by the server when presenting tags. A more descriptive name MAY be provided in the `displayName` Facet. The `val` of the `uri` MUST contain the reference location for this model or dictionary. For example,

```
<obj is=”obix:Lobby”>
{... other lobby items ...}
  <list name=”models” of=”obix:uri”>
    <uri name=”d1” displayName=”tagDict1” val=”http://example.com/tagdic”/>
  </list>
</obj>
```

One caveat to this behavior is that the presentation of the usage of a particular semantic model may divulge unwanted information about the server. For instance, a server that makes use of a medical tag dictionary and presents this in the Lobby may be undesirably advertising itself as an interesting target for individuals attempting to access confidential medical records. Therefore, it is recommended that servers SHOULD protect this section of the Lobby by only including it in communication to authenticated, authorized clients.

5.4.2 Encodings

Servers SHOULD include the encodings supported in the `encodings` Lobby Object. This is a `list` of `uris`. The name of each `uri` MUST be the MIME type of the encoding. The `val` of the `uri` SHOULD be a reference to the encoding specification. A more friendly name MAY be provided in the `displayName` attribute.

The discovery of which encoding to use for communication between a client and a server is a function of the specific binding used. Clients and servers MUST be able to support negotiation of the encoding to be used according to the binding’s error message rules. Clients SHOULD first attempt to request communication using the desired encoding, and then fall back to other encodings as required based on the encodings supported by the server.

For example, a server that supports both XML and JSON encoding as defined in the **OBIX Encodings** specification would have a Lobby that appeared as follows (note the `displayName`s used are optional):

```
<obj is=”obix:Lobby”>
{... other lobby items ...}
```



```
  <list name=”encodings” of=”obix:uri”>
    <uri name=”text/xml” displayName=”XML” val=”http://docs.oasis-open.org/obix/OBIX-Encodings/v1.0/csd01/OBIX-Encodings-v1.0-csd01.doc”/>
    <uri name=”application/json” displayName=”JSON” val=”http://docs.oasis-open.org/obix/OBIX-Encodings/v1.0/csd01/OBIX-Encodings-v1.0-csd01.doc”/>
  </list>
</obj>
```

A server that receives a request for an encoding that is not supported MUST send an UnsupportedErr response (see Section 10.2).

5.4.3 Bindings

Servers SHOULD include the available bindings supported in the `bindings` Lobby Object. This is a `list` of `uris`. The name of each `uri` SHOULD be the name of the binding as described by its corresponding specification document. The `val` of the `uri` SHOULD be a reference to the binding specification.

Servers that support multiple bindings and encodings MAY support only certain combinations of the available bindings and encodings. For example, a server may support XML encoding over the HTTP and SOAP bindings, but support JSON encoding only over the HTTP binding.

A server that receives a request for a binding/encoding pair that is not supported MUST send an `UnsupportedErr` response (see Section 10.2).

For example, a server that supports the SOAP and HTTP bindings as defined in the OBIX REST and OBIX SOAP specifications would have a Lobby that appeared as follows (note the `displayNames` used are optional):

```
<obj is=”obix:Lobby”>
{... other lobby items ...}
  <list name=”bindings” of=”obix:uri”>
    <uri name=”http” displayName=”HTTP Binding” val=” http://docs.oasis-open.org/obix/OBIX-REST/v1.0/csd01/OBIX-REST-v1.0-csd01.doc”/>
    <uri name=”soap” displayName=”SOAP Binding” val=” http://docs.oasis-open.org/obix/OBIX-REST/v1.0/csd01/OBIX-REST-v1.0-csd01.doc”/>
  </list>
</obj>
```

5.4.4 Versioning [non-normative]

Each of the subsequent subsections describes a set of `uris` that describe specifications to which a server is implemented. These specifications are expected to change over time, and the server implementation may not be updated at the same pace. Therefore, a server implementation MAY wish to provide versioning information with the `uris` that describes the date on which the specification was retrieved. This information SHOULD be included as a child element of the `uri`. It may be in the form of an `abstime` reflecting the retrieval date, or a `str` reflecting the version information. For example:

```
<obj is=”obix:Lobby”>
{... other lobby items ...}
  <list name=”bindings” of=”obix:uri”>
    <uri name=”http” displayName=”HTTP Binding” val=”http://docs.oasis-open.org/obix/OBIX-REST/v1.0/csd01/OBIX-REST-v1.0-csd01.doc”>
      <abstime name=”fetchedOn” val=”2013-11-26T3:14:15.926Z”/>
    </uri>
    <uri name=”myBinding” diaplayName=”My New Binding” val=http://example.com/my-new-binding.doc>
      <str name=”version” val=”1.2.34”/>
    </uri>
  </list>
</obj>
```


6 Naming

All OBIX objects have two potential identifiers: name and href. Name is used to define the role of an Object within its parent. Names are programmatic identifiers only; the `displayName` Facet SHOULD be used for human interaction. Naming convention is to use camel case with the first character in lowercase. The primary purpose of names is to attach semantics to sub-objects. Names are also used to indicate overrides from a Contract. A good analogy to names is the field/method names of a class in Java or C#.

Hrefs are used to attach URIs to objects. An href is always a *URI reference*, which means it might be a relative URI that requires normalization against a base URI. The exception to this rule is the href of the root Object in an OBIX document – this href MUST be an absolute URI, not a URI reference. This allows the root Object's href to be used as the effective base URI (xml:base) for normalization. A good analogy is hrefs in HTML or XLink.

Some Objects may have both a name and an href, just a name, just an href, or neither. It is common for objects within a list to not use names, since most lists are unnamed sequences of objects. The OBIX specification makes a clear distinction between names and hrefs - clients MUST NOT assume any relationship between names and hrefs. From a practical perspective many vendors will likely build an href structure that mimics the name structure, but client software MUST never assume such a relationship.

6.1 Name

The name of an Object is represented using the `name` attribute. Names are programmatic identifiers with restrictions on their valid character set. A name SHOULD contain only ASCII letters, digits, underbar, or dollar signs. A digit MUST NOT be used as the first character. Names SHOULD use lower Camel case per **Casing** with the first character in lower case, as in the examples "foo", "fooBar", "thisIsOneLongName". Within a given Object, all of its direct children MUST have unique names. Objects which don't have a `name` attribute are called *unnamed Objects*. The root Object of an OBIX document SHOULD NOT specify a `name` attribute (but almost always has an absolute href URI).

6.2 Href

The href of an Object is represented using the `href` attribute. If specified, the root Object MUST have an absolute URI. All other hrefs within an OBIX document are treated as URI references which may be relative. Because the root href is always an absolute URI, it may be used as the base for normalizing relative URIs within the OBIX document. The formal rules for URI syntax and normalization are defined in **RFC3986**. OBIX implementations MUST follow these rules. We consider a few common cases that serve as design patterns within OBIX in Section 6.3.

As a general rule every Object accessible for a read MUST specify a URI. An OBIX document returned from a read request MUST specify a root URI. However, there are certain cases where the Object is transient, such as a computed Object from an operation invocation. In these cases there MAY not be a root URI, meaning there is no way to retrieve this particular Object again. If no root URI is provided, then the server's authority URI is implied to be the base URI for resolving relative URI references.

6.3 URI Normalization

Vendors are free to use any URI scheme, although the recommendation is to use URIs since they have well defined normalization semantics. This section provides a summary of how URI normalization should work within OBIX client agents. The general rules are:

- If the URI starts with "*scheme*:" then it is a globally absolute URI
- If the URI starts with a single slash, then it is a server absolute URI
- If the URI starts with a "#", then it is a fragment identifier (discussed in next section)
- If the URI starts with "../", then the path must backup from the base

Otherwise the URI is assumed to be a relative path from the base URI

Some examples:

```
http://server/a    +  http://overthere/x  →  http://overthere/x
http://server/a    +  /x/y/z              →  http://server/x/y/z
http://server/a/b  +  c                   →  http://server/a/c
http://server/a/b/ +  c                   →  http://server/a/b/c
http://server/a/b  +  c/d                 →  http://server/a/c/d
http://server/a/b/ +  c/d                 →  http://server/a/b/c/d
http://server/a/b  +  ../c                →  http://server/c
http://server/a/b/ +  ../c                →  http://server/a/c
```

Perhaps one of the trickiest issues is whether the base URI ends with a slash. If the base URI doesn't end with a slash, then a relative URI is assumed to be relative to the base's parent (to match HTML). If the base URI does end in a slash, then relative URIs can just be appended to the base. In practice, systems organized into hierarchical URIs SHOULD always specify the base URI with a trailing slash. Retrieval with and without the trailing slash SHOULD be supported with the resulting OBIX document always adding the implicit trailing slash in the root Object's `href`.

6.4 Fragment URIs

It is not uncommon to reference an Object internal to an OBIX document. This is achieved using fragment URI references starting with the "#". Let's consider the example:

```
<obj href="http://server/whatever/">
  <enum name="switch1" range="#onOff" val="on"/>
  <enum name="switch2" range="#onOff" val="off"/>
  <list is="obix:Range" href="onOff">
    <obj name="on"/>
    <obj name="off"/>
  </list>
</obj>
```

In this example there are two Objects with a `range` Facet referencing a fragment URI. Any URI reference starting with "#" MUST be assumed to reference an Object within the same OBIX document. Clients SHOULD NOT perform another URI retrieval to dereference the Object. In this case the Object being referenced is identified via the `href` attribute.

In the example above the Object with an href of "onOff" is both the target of the fragment URI, but also has the absolute URI "http://server/whatever/onOff". But suppose we had an Object that was the target of a fragment URI within the document, but could not be directly addressed using an absolute URI? In that case the href attribute SHOULD be a fragment identifier itself. When an href attribute starts with "#" that means the only place it can be used is within the document itself:

```
…
  <list is="obix:Range" href="#onOff">
…
```


7 Contracts

OBIX Contracts are used to define inheritance in OBIX Objects. A Contract is a template, defined as an OBIX Object, that is referenced by other Objects. These templates are referenced using the `is` attribute. Contracts solve several important problems in OBIX:

Semantics	Contracts are used to define “types” within OBIX. This lets us collectively agree on common Object definitions to provide consistent semantics across vendor implementations. For example the `Alarm` Contract ensures that client software can extract normalized alarm information from any vendor’s system using the exact same Object structure.
Defaults	Contracts also provide a convenient mechanism to specify default values. Note that when serializing Object trees to XML (especially over a network), we typically don’t allow defaults to be used in order to keep client processing simple.
Type Export	It is likely that many vendors will have a system built using a statically typed language like Java or C#. Contracts provide a standard mechanism to export type information in a format that all OBIX clients can consume.

Table 7-1. Problems addressed by Contracts.

The benefit of the Contract design is its flexibility and simplicity. Conceptually Contracts provide an elegant model for solving many different problems with one abstraction. We can define new abstractions using the OBIX syntax itself. Contracts also give us a machine readable format that clients already know how to retrieve and parse –the exact same syntax is used to represent both a class and an instance.

7.1 Contract Terminology

Common terms that are useful for discussing Contracts are defined in the following Table.

Term	Definition
Contract	Contracts are the templates or prototypes used as the foundation of the OBIX type system. They may contain both syntactical and semantic behaviors.
Contract Definition	A reusable Object definition expressed as a standard OBIX Object.
Contract List	A list of one or more URIs to Contract Objects. It is used as the value of the `is`, `of`, `in` and `out` attributes. The list of URIs is separated by the space character. You can think of a Contract List as a type declaration.
Implements	When an Object specifies a Contract in its Contract List, the Object is said to *implement* the Contract. This means that the Object is inheriting both the structure and semantics of the specified Contract.
Implementation	An Object which implements a Contract is said to be an *implementation* of that Contract.

Table 7-2. Contract terminology.

7.2 Contract List

The syntax of a Contract List attribute is a list of URI references to other OBIX Objects. It is used as the value of the `is`, `of`, `in` and `out` attributes. The URIs within the list are separated by the space character (Unicode 0x20). Just like the `href` attribute, a Contract URI can be an absolute URI, server relative, or even a fragment reference. The URIs within a Contract List may be scoped with an XML namespace prefix (see “Namespace Prefixes in Contract Lists” in the **OBIX Encodings** document).

7.3 Is Attribute

An Object defines the Contracts it implements via the `is` attribute. The value of the `is` attribute is a Contract List. If the `is` attribute is unspecified, then the following rules are used to determine the implied Contract List:

- If the Object is an item inside a `list` or `feed`, then the Contract List specified by the `of` attribute is used.
- If the Object overrides (by name) an Object specified in one of its Contracts, then the Contract List of the overridden Object is used.
- If all the above rules fail, then the respective primitive Contract is used. For example, an `obj` element has an implied Contract of `obix:obj` and `real` an implied Contract of `obix:real`.

Note that element names such as `bool`, `int`, or `str` are abbreviations for implied Contracts. However if an Object implements one of the primitive types, then it MUST use the correct OBIX type name. For example if an Object implements `obix:int`, then it MUST be expressed as `<int/>`, rather than `<obj is="obix:int"/>`. Therefore it is invalid to implement multiple value types - such as implementing both `obix:bool` and `obix:int`.

7.4 Contract Inheritance

7.4.1 Structure vs Semantics

Contracts are a mechanism of inheritance – they establish the classic “is a” relationship. In the abstract sense a Contract allows us to inherit a *type*. We can further distinguish between the explicit and implicit Contract:

Explicit Contract	Defines an object structure which all implementations must conform with. This can be evaluated quantitatively by examining the Object data structure.
Implicit Contract	Defines semantics associated with the Contract. The implicit Contract is typically documented using natural language prose. It is qualitatively interpreted, rather than quantitatively interpreted.

Table 7-3. Explicit and Implicit Contracts.

For example when we say an Object implements the `Alarm` Contract, we immediately know that will have a child called `timestamp`. This structure is in the explicit contract of `Alarm` and is formally defined in its encoded definition. But we also attach semantics to what it means to be an `Alarm` Object: that the Object is providing information about an alarm event. These subjective concepts cannot be captured in machine language; rather they can only be captured in prose.

When an Object declares itself to implement a Contract it MUST meet both the explicit Contract and the implicit Contract. An Object MUST NOT put `obix:Alarm` in its Contract List unless it really represents an alarm event. There isn’t much more to say about implicit Contracts other than it is recommended that a human brain be involved. So now let’s look at the rules governing the explicit Contract.

7.4.2 Overriding Defaults

A Contract’s named children Objects are automatically applied to implementations. An implementation may choose to *override* or *default* each of its Contract’s children. If the implementation omits the child, then it is assumed to default to the Contract’s value. If the implementation declares the child (by name), then it is overridden and the implementation’s value should be used. Let’s look at an example:

```
<obj href="/def/television">
```



```
  <bool name="power"   val="false"/>
  <int  name="channel" val="2" min="2" max="200"/>
</obj>

<obj href="/livingRoom/tv" is="/def/television">
  <int name="channel" val="8"/>
  <int name="volume"  val="22"/>
</obj>
```

In this example we have a Contract Object identified with the URI “/def/television”. It has two children to store power and channel. Then we specify a living room TV instance that includes “/def/television” in its Contract List via the `is` attribute. In this Object, channel is *overridden* to 8 from its default value of 2. However since power was omitted, it is implied to *default* to false.

An override is always matched to its Contract via the `name` attribute. In the example above we knew we were overriding channel, because we declared an Object with a name of “channel”. We also declared an Object with a name of “volume”. Since volume wasn’t declared in the Contract, we assume it’s a new definition specific to this Object.

7.4.3 Attributes and Facets

Also note that the Contract’s channel Object declares a `min` and `max` Facet. These two Facets are also inherited by the implementation. Almost all attributes are inherited from their Contract including Facets, `val`, `of`, `in`, and `out`. The `href` attribute is never inherited. The `null` attribute inherits as follows:

1. If the `null` attribute is specified, then its explicit value is used;
2. If a `val` attribute is specified and `null` is unspecified, then `null` is implied to be false;
3. If neither a `val` attribute or a `null` attribute is specified, then the `null` attribute is inherited from the Contract;
4. If the `null` attribute is specified and is true, then the `val` attribute is ignored.

This allows us to implicitly override a null Object to non-null without specifying the `null` attribute.

7.5 Override Rules

Contract overrides are required to obey the implicit and explicit Contract. Implicit means that the implementation Object provides the same semantics as the Contract it implements. In the example above it would be incorrect to override channel to store picture brightness. That would break the semantic Contract.

Overriding the explicit Contract means to override the value, Facets, or Contract List. However we can never override the Object to be an incompatible value type. For example if the Contract specifies a child as `real`, then all implementations must use `real` for that child. As a special case, `obj` may be narrowed to any other element type.

We also have to be careful when overriding attributes to never break restrictions the Contract has defined. Technically this means we can *specialize* or *narrow* the value space of a Contract, but never *generalize* or *widen* it. This concept is called *covariance*. Let’s take our example from above:

```
<int name="channel" val="2" min="2" max="200"/>
```

In this example the Contract has declared a value space of 2 to 200. Any implementation of this Contract must meet this restriction. For example it would an error to override `min` to –100 since that would widen the value space. However we can narrow the value space by overriding `min` to a number greater than 2 or by overriding `max` to a number less than 200. The specific override rules applicable to each Facet are documented in section 4.1.2.

7.6 Multiple Inheritance

An Object’s Contract List may specify multiple Contract URIs to implement. This is actually quite common - even required in many cases. There are two topics associated with the implementation of multiple Contracts:

Flattening	Contract Lists SHOULD always be *flattened* when specified. This comes into play when a Contract has its own Contract List (Section 7.6.1).
Mixins	The mixin design specifies the exact rules for how multiple Contracts are merged together. This section also specifies how conflicts are handled when multiple Contracts contain children with the same name (Section 7.6.2).

Table 7-4. Contract inheritance.

7.6.1 Flattening

It is common for Contract Objects themselves to implement Contracts, just like it is common in OO languages to chain the inheritance hierarchy. However due to the nature of accessing OBIX documents over a network, we wish to minimize round trip network requests which might be required to “learn” about a complex Contract hierarchy. Consider this example:

```
<obj href="/A" />
<obj href="/B" is="/A" />
<obj href="/C" is="/B" />
<obj href="/D" is="/C" />
```

In this example if we were reading Object D for the first time, it would take three more requests to fully learn what Contracts are implemented (one for C, B, and A). Furthermore, if our client was just looking for Objects that implemented B, it would difficult to determine this just by looking at D.

Because of these issues, servers are REQUIRED to flatten their Contract inheritance hierarchy into a list when specifying the `is`, `of`, `in`, or `out` attributes. In the example above, the correct representation would be:

```
<obj href="/A" />
<obj href="/B" is="/A" />
<obj href="/C" is="/B /A" />
<obj href="/D" is="/C /B /A" />
```

This allows clients to quickly scan D’s Contract List to see that D implements C, B, and A without further requests.

Because complex servers often have a complex Contract hierarchy of Object types, the requirement to flatten the Contract hierarchy can lead to a verbose Contract List. Often many of these Contracts are from the same namespace. For example:

```
<obj name=”VSD1” href=”acme:VSD-1” is=”acmeObixLibrary:VerySpecificDevice1
acmeObixLibrary:VerySpecificDeviceBase acmeObixLibrary:SpecificDeviceType
acmeObixLibrary:BaseDevice acmeObixLibrary:BaseObject”/>
```

To save space, servers MAY choose to combine the Contracts from the same namespace and present the Contract List with the namespace followed by a colon, then a brace-enclosed list of Contract names:

```
<real name=”writableReal” is=”obix:{Point WritablePoint}”/>

<obj name=”VSD1” href=”acme:VSD-1” is=”acmeObixLibrary:{VerySpecificDevice1
VerySpecificDeviceBase SpecificDeviceType BaseDevice BaseObject}”/>
```

Clients MUST be able to consume this form of the Contract List and expand it to the standard form.

7.6.2 Mixins

Flattening is not the only reason a Contract List might contain multiple Contract URIs. OBIX also supports the more traditional notion of multiple inheritance using a mixin metaphor. Consider the following example:

```
<obj href="acme:Device">
  <str name="serialNo"/>
</obj>

<obj href="acme:Clock" is="acme:Device">
  <op name="snooze"/>
  <int name="volume" val="0"/>
</obj>

```



```
<obj href="acme:Radio" is="acme:Device ">
  <real name="station" min="87.0" max="107.5"/>
  <int name="volume" val="5"/>
</obj>

<obj href="acme:ClockRadio" is="acme:Radio acme:Clock acme:Device"/>
```

In this example `ClockRadio` implements both `Clock` and `Radio`. Via flattening of `Clock` and `Radio`, `ClockRadio` also implements `Device`. In OBIX this is called a *mixin* – `Clock`, `Radio`, and `Device` are mixed into (merged into) `ClockRadio`. Therefore `ClockRadio` inherits four children: `serialNo`, `snooze`, `volume`, and `station`. Mixins are a form of multiple inheritance akin to Java/C# interfaces (remember OBIX is about the type inheritance, not implementation inheritance).

Note that `Clock` and `Radio` both implement `Device`. This inheritance pattern where two types both inherit from a base, and are themselves both inherited by a single type, is called a "diamond" pattern from the shape it takes when the class hierarchy is diagrammed. From `Device`, `ClockRadio` inherits a child named `serialNo`. Furthermore notice that both `Clock` and `Radio` declare a child named `volume`. This naming collision could potentially create confusion for what `serialNo` and `volume` mean in `ClockRadio`.

In OBIX we solve this problem by flattening the Contract's children using the following rules:

1. Process the Contract definitions in the order they are listed
2. If a new child is discovered, it is mixed into the Object's definition
3. If a child is discovered we already processed via a previous Contract definition, then the previous definition takes precedence. However it is an error if the duplicate child is not *Contract compatible* with the previous definition (see Section 7.7).

In the example above this means that `Radio.volume` is the definition we use for `ClockRadio.volume`, because `Radio` has a higher precedence than `Clock` (it is first in the Contract List). Thus `ClockRadio.volume` has a default value of "5". However it would be invalid if `Clock.volume` were declared as `str`, since it would not be Contract compatible with `Radio`'s definition as an `int` – in that case `ClockRadio` could not implement both `Clock` and `Radio`. It is the server vendor's responsibility not to create incompatible name collisions in Contracts.

The first Contract in a list is given specific significance since its definition trumps all others. In OBIX this Contract is called the *Primary Contract*. It is recommended that the Primary Contract implement all the other Contracts specified in the Contract List (this actually happens quite naturally by itself in many programming languages). This makes it easier for clients to bind the Object into a strongly typed class if desired. Contracts MUST NOT implement themselves nor have circular inheritance dependencies.

7.7 Contract Compatibility

A Contract List which is covariantly substitutable with another Contract List is said to be *Contract compatible*. Contract compatibility is a useful term when talking about mixin rules and overrides for lists and operations. It is a fairly common sense notion similar to previously defined override rules – however, instead of the rules applied to individual Facet attributes, we apply it to an entire Contract List.

A Contract List X is compatible with Contract List Y, if and only if X narrows the value space defined by Y. This means that X can narrow the set of Objects which implement Y, but never expand the set. Contract compatibility is not commutative (X is compatible with Y does not imply Y is compatible with X). Practically, this can be expressed as: X can add new URIs to Y's list, but never take any away.

7.8 Lists and Feeds

Implementations derived from `list` or `feed` Contracts inherit the `of` attribute. Like other attributes we can override the `of` attribute, but only if Contract compatible - a server SHOULD include all of the URIs in the Contract's `of` attribute, but it MAY add additional ones (see Section 7.7).

Lists and feeds also have the special ability to implicitly define the Contract List of their contents. In the following example it is implied that each child element has a Contract List of `/def/MissingPerson` without actually specifying the `is` attribute in each list item:

```
<list of="/def/MissingPerson">
  <obj> <str name="fullName" val="Jack Shephard"/> </obj>
  <obj> <str name="fullName" val="John Locke"/> </obj>
  <obj> <str name="fullName" val="Kate Austen"/> </obj>
</list>
```

If an element in the list or feed does specify its own `is` attribute, then it MUST be Contract compatible with the `of` attribute.

If an implementer wishes to specify that a list should contain references to a given type, then the server SHOULD include `obix:ref` in the `of` attribute. This MUST be the first URI in the `of` attribute. For example, to specify that a list should contain references to obix:History Objects (as opposed to inline History Objects):

```
<list name="histories" of="obix:ref obix:History"/>
```

In many cases a server will implement its own management of the URI scheme of the child elements of a `list`. For example, the `href` attribute of child elements may be a database key, or some other string defined by the server when the child is added. Servers will not, in general, allow clients to specify this URI during addition of child elements through a direct write to a list's subordinate URI.

Therefore, in order to add child elements to a list which supports client addition of list elements, servers MUST support adding list elements by writing to the `list` URI with an Object of a type that matches the list's Contract. Servers MUST return the written resource (including any server-assigned `href`) upon successful completion of the write.

For example, given a `list` of <real> elements, and presupposing a server-imposed URI scheme:

```
<list href=”/a/b” of=”obix:real” writable=”true”/>
```

Writing to the list URI itself will replace the entire list if the server supports this behavior:

WRITE /a/b

```
<list of=”obix:real”>
 <real name=”foo” val=”10.0”/>
 <real name=”bar” val=”20.0”/>
</list>
```

returns:

```
<list href=”/a/b” of=”obix:real”>
 <real name=”foo” href=”1” val=”10.0”/>
 <real name=”bar” href=”2” val=”20.0”/>
</list>
```

Writing a single element of type <real> will add this element to the list.

WRITE /a/b

```
<real name=”baz” val=”30.0”/>
```

returns:

```
<real name=”baz” href=”/a/b/3” val=”30.0”/>
```

while the list itself is now:

```
<list href=”/a/b” of=”obix:real”>
 <real name=”foo” href=”1” val=”10.0”/>
 <real name=”bar” href=”2” val=”20.0”/>
 <real name=”baz” href=”3” val=”30.0”/>
</list>
```

Note that if a client has the correct URI to reference a list child element, this can still be used to modify the value of the element directly:

WRITE /a/b/3

```
<real name=”baz2” val=”33.0”/>
```

returns:

```
<real name=”baz2” href=”/a/b/3” val=”33.0”/>
```


and the list has been modified to:

```
<list href=”/a/b” of=”obix:real”>
 <real name=”foo” href=”1” val=”10.0”/>
 <real name=”bar” href=”2” val=”20.0”/>
 <real name=”baz” href=”3” val=”33.0”/>
</list>
```


8 Operations

OBIX Operations are the exposed actions that an OBIX Object can be commanded to take, i.e., they are things you can invoke to “do” something to the Object. Typically object-oriented languages express this concept as the publicly accessible methods on the object. They generally map to commands rather than a variable that has continuous state. Unlike Value Objects which represent an Object and its current state, the `op` element merely represents the definition of an operation you can invoke.

All operations take exactly one Object as a parameter and return exactly one Object as a result. The `in` and `out` attributes define the Contract List for the input and output Objects. If you need multiple input or output parameters, then wrap them in a single Object using a Contract as the signature. For example:

```
<op href="/addTwoReals" in="/def/AddIn" out="obix:real"/>

<obj href="/def/AddIn">
  <real name="a"/>
  <real name="b"/>
</obj>
```

Objects can override the operation definition from one of their Contracts. However the new `in` or `out` Contract List MUST be Contract compatible (see Section 7.7) with the Contract’s definition.

If an operation doesn’t require a parameter, then specify `in` as `obix:Nil`. If an operation doesn’t return anything, then specify `out` as `obix:Nil`. Occasionally an operation is inherited from a Contract which is unsupported in the implementation. In this case set the `status` attribute to `disabled`.

Operations are always invoked via their own `href` attribute (not their parent’s `href`). Therefore operations SHOULD always specify an `href` attribute if you wish clients to invoke them. A common exception to this rule is Contract definitions themselves.

9 Object Composition

Object Composition describes how multiple OBIX Objects representing individual pieces are combined to form a larger unit. The individual pieces can be as small as the various data fields in a simple thermostat, as described in Section 2, or as large as entire buildings, each themselves composed of multiple networks of devices. All of the OBIX Objects are linked together via URIs, similar to the way that the World Wide Web is a group of HTML documents hyperlinked together through URIs These OBIX Objects may be static documents like Contracts or device descriptions. Or they may be real-time data or services.

Individual Objects are composed together in two ways to define this web. Objects may be composed together via *containment* or via *reference*.

9.1 Containment

Any OBIX Object may contain zero or more children Objects. This even includes Objects which might be considered primitives such as `bool` or `int`. All Objects are open ended and free to specify new Objects which may not be in the Object's Contract. Containment is represented in the XML syntax by nesting the XML elements:

```
<obj href="/a/">
  <list name="b" href="b">
    <obj href="b/c"/>
  </list>
</obj>
```

In this example the Object identified by "/a" contains "/a/b", which in turn contains "/a/b/c". Child Objects may be named or unnamed depending on if the `name` attribute is specified (Section 6.1). In the example, "/a/b" is named and "/a/b/c" is unnamed. Typically named children are used to represent fields in a record, structure, or class type. Unnamed children are often used in lists.

9.2 References

To discuss references, let's return to our World Wide Web metaphor. Although the WWW is a web of individual HTML elements like `<p>` and `<div>`, we don't actually pass individual <p> elements around over the network. Rather we "chunk" them into HTML documents and always pass the entire document over the network. To tie it all together, we create links between documents using the `<a>` anchor element. These anchors serve as place holders, referencing outside documents via a URI.

An OBIX reference is basically just like an HTML anchor. It serves as placeholder to "link" to another OBIX Object via a URI. While containment is best used to model small trees of data, references may be used to model very large trees or graphs of Objects. With references we can link together all OBIX Objects on the Internet to create the OBIX Web.

As a clue to clients consuming OBIX references, the server SHOULD specify the type of the referenced Object using the `is` attribute. In addition, for the `list` element type, the server SHOULD use the `of` attribute to specify the type of Objects contained by the `list`. This allows the client to prepare the proper visualizations, data structures, etc. for consuming the Object when it accesses the actual Object. For example, a server might provide a reference to a list of available points:

```
<ref name="points" is="obix:list" of="obix:Point"/>
```

9.3 Extents

Within any problem domain, the intra-model relationships can be expressed by using either containment or references. The choice changes the semantics of both the model expression as well as the method for accessing the elements within the model. The containment relationship is imbued with special semantics regarding encoding and event management. If the model is expressed through containment, then we use the term *Extent* to refer to the tree of children contained within that Object, down to references. Only Objects which have an href have an Extent. Objects without an href are always included within the Extent of one or more referenceable Objects which we term its *Ancestors*. This is demonstrated in the following example.

```
<obj href="/a/">
  <obj name="b" href="b">
    <obj name="c"/>
    <ref name="d" href="/d"/>
  </obj>
  <ref name="e" href="/e"/>
</obj>
```

In the example above, we have five Objects named ‘a’ to ‘e’. Because ‘a’ includes an href, it has an associated extent, which encompasses ‘b’ and ‘c’ by containment and ‘d’ and ‘e’ by reference. Likewise, ‘b’ has an href which results in an extent encompassing ‘c’ by containment and ‘d’ by reference. Object ‘c’ does not provide a direct href, but exists in both the ‘a’ and ‘b’ Objects’ extents. Note an Object with an href has exactly one extent, but can be nested inside multiple extents.

9.3.1 Inlining Extents

When marshaling Objects into an OBIX document, it is required that an extent always be fully inlined into the document. The only valid Objects which may be references outside the document are `ref` Objects. In order to allow conservation of bandwidth usage, processing time, and storage requirements, servers SHOULD use non-`ref` Objects only for representing primitive children which have no further extent. `Ref`s SHOULD be used for all complex children that have further structure under them. Clients MUST be able to consume the `ref`s and then request the referenced object if it is needed for the application. As an example, consider a server which has the following object tree, represented here with full extent:

```
<obj name=”MyBuilding” href=”/building/”>
  <str name=”address” val=”123 Main Street”/>
  <obj name=”Floor1”>
    <obj name=”Zone1”>
      <obj name=”Room1”/>
    </obj>
  </obj>
</obj>
```

When marshaled into an OBIX document to respond to a client Read request of the /building/ URI, the server SHOULD inline only the address, and use a `ref` for Floor1:

```
<obj name=”MyBuilding” href=”/building/”>
  <str name=”address” val=”123 Main Street”/>
  <ref name=”Floor1” href=”floor1”/>
</obj>
```

If the Object implements a Contract, then it is required that the extent defined by the Contract be fully inlined into the document (unless the Contract itself defined a child as a `ref` element). An example of a Contract which specifies a child as a `ref` is `Lobby.about` (Section 5.1).

9.4 Alternate Hierarchies

An OBIX Server MAY present *Tags* that reference additional information about each OBIX Object. If these Tags are part of a formal semantic model, e.g., Haystack, BIM, etc., then the Tags will be identified by reference to its source semantic model. The identifier for such Tags, along with the URI for the semantic model it represents, MUST be declared in the Lobby (see Section 5 for a description of the Lobby Object). A server MUST use the semicolon character (**;**) to indicate an alternate hierarchy. For example, a server might present tag metadata from tag dictionary d1 in presenting a particular object in its system:

```
<real href=”/bldg/floor1/room101/” name=”Room101” val=”70.0”>
 <ref name=”tags” href=”../room101;meta”/>
</real>

<obj name=”tags” href=”/bldg/floor1/room101;meta”>
  <obj name=”d1:temperature”/>
  <int name=”d1:roomNumber” val=”101”/>
  <uri name=”d1:vavReference” val=”/bldg/vavs/vav101”/>
 </obj>
```


Servers SHOULD only provide this information to clients that are properly authenticated and authorized, to avoid providing a vector for attack if usage of a particular model identifies the server as an interesting target.

The metadata SHOULD be presented using the `ref` element, so this additional information can be skipped during normal encoding. If a client is able to consume the metadata, it SHOULD ask for the metadata by requesting the metadata hierarchy.

OBIX Clients SHALL ignore information that they do not understand. In particular, a conformant client that is presented with Tags that it does not understand MUST ignore those Tags. No OBIX Server may require understanding of these Tags for interoperation.

10 Networking

The heart of OBIX is its object model and associated encoding. However, the primary use case for OBIX is to access information and services over a network. The OBIX architecture is based on a client/server network model, described below:

Server	An entity containing OBIX enabled data and services. Servers respond to requests from client over a network.
Client	An entity which makes requests to servers over a network to access OBIX enabled data and services.

Table 10-1. Network model for OBIX.

There is nothing to prevent a device or system from being both an OBIX client and server. However, a key tenet of OBIX is that a client is NOT REQUIRED to implement server functionality which might require a server socket to accept incoming requests.

10.1 Service Requests

All service requests made against an OBIX server can be distilled to 4 atomic operations, expressed in the following Table:

Request	Description
Read	Return the current state of an object at a given URI as an OBIX Object.
Write	Update the state of an existing object at a URI. The state to write is passed over the network as an OBIX Object. The new updated state is returned in an OBIX Object.
Invoke	Invoke an operation identified by a given URI. The input parameter and output result are passed over the network as an OBIX Object.
Delete	Delete the object at a given URI.

Table 10-2. OBIX Service Requests.

Exactly how these requests and responses are implemented between a client and server is called a *protocol binding*. The OBIX specification defines standard protocol bindings in separate companion documents. All protocol bindings MUST follow the same read, write, invoke, and delete semantics discussed next.

10.1.1 Read

The read request specifies an object's URI and the read response returns the current state of the object as an OBIX document. The response MUST include the Object's complete extent (see Section 9.3). Servers may return an `err` Object to indicate the read was unsuccessful – the most common error is `obix:BadUriErr` (see Section 10.2 for standard error Contracts).

10.1.2 Write

The write request is designed to overwrite the current state of an existing Object. The write request specifies the URI of an existing Object and its new desired state. The response returns the updated state of the Object. If the write is successful, the response MUST include the Object's complete extent (see Section 9.3). If the write is unsuccessful, then the server MUST return an `err` Object indicating the failure.

The server is free to completely or partially ignore the write, so clients SHOULD be prepared to examine the response to check if the write was successful. Servers may also return an `err` Object to indicate the write was unsuccessful.

Clients are not required to include the Object's full extent in the request. Objects explicitly specified in the request object tree SHOULD be overwritten or "overlaid" over the server's actual object tree. Only the `val` attribute should be specified for a write request (outside of identification attributes such as `name`). The `null` attribute MAY also be used to set an Object to null. If the `null` attribute is not specified and the `val` attribute is specified, then it is implied that null is false. A write operation that provides Facets has unspecified behavior. When writing `int` or `reals` with `units`, the write value MUST be in the same units as the server specifies in read requests – clients MUST NOT provide a different `unit` Facet and expect the server to auto-convert (in fact the `unit` Facet SHOULD NOT be included in the request).

10.1.3 Invoke

The invoke request is designed to trigger an operation. The invoke request specified the URI of an `op` Object and the input argument Object. The response includes the output Object. The response MUST include the output Object's complete extent (see Section 9.3). Servers MAY instead return an `err` Object to indicate the invocation was unsuccessful.

10.1.4 Delete

The delete request is designed to remove an existing Object from the server. The delete request specifies the URI of an existing Object. If the delete is successful, the server MUST return an empty response. If the delete is unsuccessful, the server MUST return an `err` Object indicating the failure.

10.2 Errors

Request errors are conveyed to clients with the `err` element. Any time an OBIX server successfully receives a request and the request cannot be processed, then the server SHOULD return an `err` Object to the client. Returning a valid OBIX document with `err` SHOULD be used when feasible rather than protocol specific error handling (such as an HTTP response code). Such a design allows for consistency with batch request partial failures and makes protocol binding more pluggable by separating data transport from application level error handling.

The following Table describes the base Contracts predefined for representing common errors:

Err Contract	Usage
BadUriErr	Used to indicate either a malformed URI or a unknown URI
UnsupportedErr	Used to indicate an a request which isn't supported by the server implementation (such as an operation defined in a Contract, which the server doesn't support)
PermissionErr	Used to indicate that the client lacks the necessary security permission to access the object or operation

Table 10-3. OBIX Error Contracts.

The Contracts for these errors are:

```
<err href="obix:BadUriErr"/>
<err href="obix:UnsupportedErr"/>
<err href="obix:PermissionErr"/>
```

If one of the above Contracts makes sense for an error, then it SHOULD be included in the `err` element's `is` attribute. It is strongly encouraged to also include a useful description of the problem in the `display` attribute.

10.3 Localization

Servers SHOULD localize appropriate data based on the desired locale of the client agent. Localization SHOULD include the `display` and `displayName` attributes. The desired locale of the client SHOULD be determined through authentication or through a mechanism appropriate to the binding used. A suggested algorithm is to check if the authenticated user has a preferred locale configured in the server's user database, and if not then fallback to the locale derived from the binding.

Localization MAY include auto-conversion of units. For example if the authenticated user has configured a preferred unit system such as English versus Metric, then the server might attempt to convert values with an associated `unit` facet to the desired unit system.

11 Core Contract Library

This chapter defines some fundamental Object Contracts that serve as building blocks for the OBIX specification.

11.1 Nil

The `obix:Nil` Contract defines a standardized null Object. Nil is commonly used for an operation's `in` or `out` attribute to denote the absence of an input or output. The definition:

```
<obj href="obix:Nil" null="true"/>
```

11.2 Range

The `obix:Range` Contract is used to define a `bool` or `enum`'s range. `Range` is a list Object that contains zero or more Objects called the range items. Each item's `name` attribute specifies the identifier used as the literal value of an `enum`. Item ids are never localized, and MUST be used only once in a given range. You may use the optional `displayName` attribute to specify a localized string to use in a user interface. The definition of `Range`:

```
<list href="obix:Range" of="obix:obj"/>
```

An example:

```
<list href="/enums/OffSlowFast" is="obix:Range">
  <obj name="off"  displayName="Off"/>
  <obj name="slow" displayName="Slow Speed"/>
  <obj name="fast" displayName="Fast Speed"/>
</list>
```

The `range` Facet may be used to define the localized text of a `bool` value using the ids of "true" and "false":

```
<list href="/enums/OnOff" is="obix:Range">
  <obj name="true"  displayName="On"/>
  <obj name="false" displayName="Off"/>
</list >
```

11.3 Weekday

The `obix:Weekday` Contract is a standardized enum for the days of the week:

```
<enum href="obix:Weekday" range="#Range">
  <list href="#Range" is="obix:Range">
    <obj name="sunday" />
    <obj name="monday" />
    <obj name="tuesday" />
    <obj name="wednesday" />
    <obj name="thursday" />
    <obj name="friday" />
    <obj name="saturday" />
  </list>
</enum>
```

11.4 Month

The `obix:Month` Contract is a standardized enum for the months of the year:

```
<enum href="obix:Month" range="#Range">
  <list href="#Range" is="obix:Range">
    <obj name="january" />
    <obj name="febuary" />
    <obj name="march" />
    <obj name="april" />
    <obj name="may" />
```



```
    <obj name="june" />
    <obj name="july" />
    <obj name="august" />
    <obj name="september" />
    <obj name="october"  />
    <obj name="november" />
    <obj name="december" />
  </list>
</enum>
```

11.5 Units

Representing units of measurement in software is a thorny issue. OBIX provides a unit framework for mathematically defining units within the object model. An extensive database of predefined units is also provided.

All units measure a specific quantity or dimension in the physical world. Most known dimensions can be expressed as a ratio of the seven fundamental dimensions: length, mass, time, temperature, electrical current, amount of substance, and luminous intensity. These seven dimensions are represented in the **SI Units** system respectively as kilogram (kg), meter (m), second (sec), Kelvin (K), ampere (A), mole (mol), and candela (cd).

The `obix:Dimension` Contract defines the ratio of the seven SI units using a positive or negative exponent:

```
<obj href="obix:Dimension">
  <int name="kg"  val="0"/>
  <int name="m"   val="0"/>
  <int name="sec" val="0"/>
  <int name="K"   val="0"/>
  <int name="A"   val="0"/>
  <int name="mol" val="0"/>
  <int name="cd"  val="0"/>
</obj>
```

A `Dimension` Object contains zero or more ratios of `kg`, `m`, `sec`, `K`, `A`, `mol`, or `cd`. Each of these ratio maps to the exponent of that base SI unit. If a ratio is missing then the default value of zero is implied. For example acceleration is m/s^2, which would be encoded in OBIX as:

```
<obj is="obix:Dimension">
  <int name="m"   val="1"/>
  <int name="sec" val="-2"/>
</obj>
```

Units with equal dimensions are considered to measure the same physical quantity. This is not always precisely true, but is good enough for practice. This means that units with the same dimension are convertible. Conversion can be expressed by specifying the formula required to convert the unit to the dimension’s normalized unit. The normalized unit for every dimension is the ratio of SI units itself. For example the normalized unit of energy is the joule $m^2 \bullet kg \bullet s^{-2}$. The kilojoule is 1000 joules and the watt-hour is 3600 joules. Most units can be mathematically converted to their normalized unit and to other units using the linear equations:

```
unit = dimension • scale + offset
toNormal = scalar • scale + offset
fromNormal = (scalar - offset) / scale
toUnit = fromUnit.fromNormal( toUnit.toNormal(scalar) )
```

There are some units which don’t fit this model including logarithm units and units dealing with angles. But this model provides a practical solution for most problem spaces. Units which don’t fit this model SHOULD use a dimension where every exponent is set to zero. Applications SHOULD NOT attempt conversions on these types of units.

The `obix:Unit` Contract defines a unit including its dimension and its toNormal equation:

```
<obj href="obix:Unit">
  <str  name="symbol"/>
  <obj  name="dimension" is="obix:Dimension"/>
  <real name="scale" val="1"/>
```



```
  <real name="offset" val="0"/>
</obj>
```

The `unit` element contains `symbol`, `dimension`, `scale`, and `offset` sub-Objects, as described in the following Table:

symbol	The `symbol` element defines a short abbreviation to use for the unit. For example "°F" would be the symbol for degrees Fahrenheit. The `symbol` element SHOULD always be specified.
dimension	The `dimension` Object defines the dimension of measurement as a ratio of the seven base SI units. If omitted, the `dimension` Object defaults to the `obix:Dimension` Contract, in which case the ratio is the zero exponent for all seven base units.
scale	The `scale` element defines the scale variable of the toNormal equation. The `scale` Object defaults to 1.
offset	The `offset` element defines the offset variable of the toNormal equation. If omitted then `offset` defaults to 0.

Table 11-1. OBIX Unit composition.

The `display` attribute SHOULD be used to provide a localized full name for the unit based on the client's locale. If the `display` attribute is omitted, clients SHOULD use `symbol` for display purposes.

An example for the predefined unit for kilowatt:

```
<obj href="obix:units/kilowatt" display="kilowatt">
  <str name="symbol" val="kW"/>
  <obj name="dimension">
    <int name="m" val="2"/>
    <int name="kg" val="1"/>
    <int name="sec" val="-3"/>
  </obj>
  <real name="scale" val="1000"/>
</obj>
```

Automatic conversion of units is considered a localization issue.

12 Watches

A key requirement of OBIX is access to real-time information. We wish to enable clients to efficiently receive access to rapidly changing data. However, we don't want to require clients to implement web servers or expose a well-known IP address. In order to address this problem, OBIX provides a model for event propagation called *Watches*.

The Implicit Contract for Watch is described in the following lifecycle:

- The client creates a new Watch Object with the `make` operation on the server's WatchService URI. The server defines a new Watch Object and provides a URI to access the new Watch.
- The client registers (and unregisters) Objects to watch using operations on the Watch Object.
- The server tracks events that occur on the Objects in the Watch.
- The client receives events from the server about changes to Objects in the Watch. The events can be polled by the client (see 12.1) or pushed by the server (see 12.2).
- The client may invoke the `pollRefresh` operation at any time to obtain a full list of the current value of each Object in the Watch.
- The Watch is freed, either by the explicit request of the client using the `delete` operation, or when the server determines the Watch is no longer being used. See Sections 12.1 and 12.2 for details on the criteria for server removal of Watches. When the Watch is freed, the Objects in it are no longer tracked by the server and the server may return any resources used for it to the system.

Watches allow a client to maintain a real-time cache of the current state of one or more Objects. They are also used to access an event stream from a `feed` Object. Watches also serve as the standardized mechanism for managing per-client state on the server via leases.

12.1 Client Polled Watches

When the underlying binding does not allow the server to send unsolicited messages, the Watch must be periodically polled by the client. The Implicit Contract for Watch in this scenario is extended as follows:

- The client SHOULD periodically poll the Watch URI using the `pollChanges` operation to obtain the events which have occurred since the last poll.
- In addition to freeing the Watch by explicit request of the client, the server MAY free the Watch if the client fails to poll for a time greater than the *lease time* of the Watch. See the `lease` property in Section 12.4.5.

12.2 Server Pushed Watches

Some bindings, for example the **OBIX WebSockets** binding, may allow unsolicited transmission by either the client or the server. If this is possible the standard Implicit Contract for Watch behavior is extended as follows:

- Change events are sent by the server directly to the client as unsolicited updates.
- The lease time property of the Watch MUST NOT be used for server automatic removal of the Watch. The Watch SHOULD remain active without the need for the client to invoke the `pollChanges` or `pollRefresh` operations.
- The Watch MUST be removed by the server upon termination of the underlying session between the client and server, in addition to the normal removal upon explicit client request.
- The server MUST return an empty list upon invocation of the `pollChanges` operation.

Watches used in servers that can push events MUST provide three additional properties for configuring the Watch behavior:

- `bufferDelay`: The implicit contract for `bufferDelay` is the period of time for which any events on watched objects will be buffered before being sent by the server in an update. Clients must be able to regulate the flow of messages from the server. A common scenario is an OBIX client application on a mobile device where the bandwidth usage is important; for example, a server sending updates every 50 milliseconds as a sensor value jitters around will cause problems. On the other hand, server devices may be constrained in terms of the available space for buffering changes. Servers are free to set a maximum value on `bufferDelay` through the `max` Facet to constrain the maximum delay before the server will report events.
- `maxBufferedEvents`: Servers may also use the `maxBufferedEvents` property to indicate the maximum number of events that can be retained before the buffer must be sent to the client to avoid missing events.
- `bufferPolicy`: This enum property defines the handling of the buffer on the server side when further events occur while the buffer is full. A value of `violate` means that the `bufferDelay` property is violated and the events are sent, allowing the buffer to be emptied. A value of `LIFO` (last-in-first-out) means that the most recently added buffer event is replaced with the new event. A value of `FIFO` (first-in-first-out) means that the oldest buffer event is dropped to make room for the new event.
- **NOTE:** A server using a `bufferPolicy` of either `LIFO` or `FIFO` will not send events when a buffer overrun occurs, and this means that some events will not be received by the client. It is up to the client and server to negotiate appropriate values for these three properties to ensure that events are not lost, if that is important to the application.

Note that `bufferDelay` MUST be writable by the client, as the client capabilities typically constrain the bandwidth usage. Server capabilities typically constrain `maxBufferedEvents`, and thus this is generally not writable by clients.

12.3 WatchService

The `WatchService` Object provides a well-known URI as the factory for creating new Watches. The `WatchService` URI is available directly from the `Lobby` Object. The Contract for `WatchService`:

```
<obj href="obix:WatchService">
  <op name="make" in="obix:Nil" out="obix:Watch"/>
</obj>
```

The make operation returns a new empty Watch Object as an output. The href of the newly created Watch Object can then be used for invoking operations to populate and poll the data set.

12.4 Watch

The `Watch` Object is used to manage a set of Objects which are subscribed by clients to receive the latest events. The Explicit Contract definitions are:

```
<obj href="obix:Watch">
  <reltime name="lease" min="PT0S" writable="true"/>
  <reltime name=”bufferDelay” min=”PT0S” writable=”true” null=”true”/>
  <int name=”maxBufferedEvents” null=”true”/>
  <enum name=”bufferPolicy” is=”obix:WatchBufferPolicy” null=”true”/>
  <op name="add"    in="obix:WatchIn" out="obix:WatchOut"/>
  <op name="remove" in="obix:WatchIn"/>
  <op name="pollChanges" out="obix:WatchOut"/>
  <op name="pollRefresh" out="obix:WatchOut"/>
  <op name="delete"/>
</obj>

<enum href=”obix:WatchBufferPolicy” range=”#Range”>
  <list href=”#Range” is=”obix:Range”>
    <obj name=”violate” />
    <obj name=”LIFO” />
    <obj name=”FIFO” />
  </list>
</enum>
```



```
<obj href="obix:WatchIn">
  <list name="hrefs" of="obix:WatchInItem"/>
</obj>

<uri href="obix:WatchInItem">
  <obj name="in"/>
</uri>

<obj href="obix:WatchOut">
  <list name="values" of="obix:obj"/>
</obj>
```

Many of the Watch operations use two Contracts: `obix:WatchIn` and `obix:WatchOut`. The client identifies Objects to `add` and `remove` from the poll list via WatchIn. This Object contains a list of URIs. Typically these URIs SHOULD be server relative.

The server responds to `add`, `pollChanges`, and `pollRefresh` operations via the WatchOut Contract. This Object contains the list of subscribed Objects - each Object MUST specify an href URI using the exact same string as the URI identified by the client in the corresponding WatchIn. Servers MUST NOT perform any case conversions or normalization on the URI passed by the client. This allows client software to use the URI string as a hash key to match up server responses.

12.4.1 Watch.add

Once a Watch has been created, the client can add new Objects to the Watch using the `add` operation. The Objects returned are required to specify an href using the exact string representation input by the client. If any Object cannot be processed, then a partial failure SHOULD be expressed by returning an `err` Object with the respective href. Subsequent URIs MUST NOT be affected by the failure of one invalid URI. The `add` operation MUST never return Objects not explicitly included in the input URIs (even if there are already existing Objects in the watch list). No guarantee is made that the order of Objects in `WatchOut` matches the order in of URIs in `WatchIn` – clients must use the URI as a key for matching.

Note that the URIs supplied via WatchIn may include an optional `in` parameter. This parameter is only used when subscribing a Watch to a `feed` Object. Feeds also differ from other Objects in that they return a list of historic events in WatchOut. Feeds are discussed in detail in Section12.6.

It is invalid to add an `op`'s href to a Watch; the server MUST report an err.

If an attempt is made to add a URI to a Watch which was previously already added, then the server SHOULD return the current Object's value in the `WatchOut` result, but treat poll operations as if the URI was only added once – polls SHOULD only return the Object once. If an attempt is made to add the same URI multiple times in the same `WatchIn` request, then the server SHOULD only return the Object once.

12.4.1.1 Watch Object URIs

The lack of a trailing slash in watched Object URIs can cause problems with Watches. Consider a client which adds a URI to a Watch without a trailing slash. The client will use this URI as a key in its local hashtable for the Watch. Therefore the server MUST use the URI exactly as the client specified. However, if the Object's extent includes child Objects they will not be able to use relative URIs. It is RECOMMENDED that servers fail fast in these cases and return a BadUriErr when clients attempt to add a URI without a trailing slash to a Watch (even though they may allow it for a normal read request).

12.4.2 Watch.remove

The client can remove Objects from the watch list using the `remove` operation. A list of URIs is input to `remove`, and the Nil Object is returned. Subsequent `pollChanges` and `pollRefresh` operations MUST cease to include the specified URIs. It is possible to remove every URI in the watch list; but this scenario MUST NOT automatically free the Watch, rather normal poll and lease rules still apply. It is invalid to use the `WatchInItem.in` parameter for a `remove` operation.

12.4.3 Watch.pollChanges

Clients SHOULD periodically poll the server using the `pollChanges` operation. This operation returns a list of the subscribed Objects which have changed. Servers SHOULD only return the Objects which have been modified since the last poll request for the specific Watch. As with `add`, every Object MUST specify an href using the exact same string representation the client passed in the original `add` operation. The entire extent of the Object SHOULD be returned to the client if any one thing inside the extent has changed on the server side.

Invalid URIs MUST never be included in the response (only in `add` and `pollRefresh`). An exception to this rule is when an Object which is valid is removed from the URI space. Servers SHOULD indicate an Object has been removed via an `err` with the `BadUriErr` Contract.

12.4.4 Watch.pollRefresh

The `pollRefresh` operation forces an update of every Object in the watch list. The server MUST return every Object and its full extent in the response using the href with the exact same string representation passed by the client in the original `add`. Invalid URIs in the poll list SHOULD be included in the response as an `err` element. A `pollRefresh` resets the poll state of every Object, so that the next `pollChanges` only returns Objects which have changed state since the `pollRefresh` invocation.

12.4.5 Watch.lease

All Watches have a *lease time*, specified by the `lease` child. If the lease time elapses without the client initiating a request on the Watch, and the Watch is a client-polled Watch, then the server MAY *expire* the Watch. Every new poll request resets the lease timer. So as long as the client polls at least as often as the lease time, the server SHOULD maintain the Watch. The following requests SHOULD reset the lease timer: read of the Watch URI itself or invocation of the `add`, `remove`, `pollChanges`, or `pollRefresh` operations.

Clients may request a different lease time by writing to the `lease` Object (requires servers to assign an href to the `lease` child). The server is free to honor the request, cap the lease within a specific range, or ignore the request. In all cases the write request will return a response containing the new lease time in effect.

Servers SHOULD report expired Watches by returning an `err` Object with the `BadUriErr` Contract. As a general principle servers SHOULD honor Watches until the lease runs out (for client-polled Watches) or the client explicitly invokes `delete`. However, servers are free to cancel Watches as needed (such as power failure) and the burden is on clients to re-establish a new Watch.

12.4.6 Watch.delete

The `delete` operation can be used to cancel an existing Watch. Clients SHOULD always delete their Watch when possible to be good OBIX citizens. However servers MUST always cleanup correctly without an explicit delete when the lease expires or the session is terminated.

12.5 Watch Depth

When a Watch is put on an Object which itself has child Objects, how does a client know how “deep” the subscription goes? OBIX requires Watch depth to match an Object‘s extent (see Section 9.3). When a Watch is put on a target Object, a server MUST notify the client of any changes to any of the Objects within that target Object’s extent. If the extent includes `feed` Objects, they are not included in the Watch – feeds have special Watch semantics discussed in Section 12.6. This means a Watch is inclusive of all descendents within the extent except `refs` and `feeds`.

12.6 Feeds

Servers may expose event streams using the `feed` Object. The event instances are typed via the feed's `of` attribute. Clients subscribe to events by adding the feed's href to a Watch, optionally passing an input parameter which is typed via the feed's `in` attribute. The Object returned from `Watch.add` is a list of historic events (or the empty list if no event history is available). Subsequent calls to `pollChanges` return the list of events which have occurred since the last poll.

Let's consider a simple example for an Object which fires an event when its geographic location changes:

```
<obj href="/car/">
  <feed href="moved" of="/def/Coordinate"/>
<obj>

<obj href="/def/Coordinate">
  <real name="lat"/>
  <real name="long"/>
</obj>
```

We subscribe to the moved event feed by adding "/car/moved" to a Watch. The WatchOut will include the list of any historic events which have occurred up to this point in time. If the server does not maintain an event history this list will be empty:

```
<obj is="obix:WatchIn">
  <list names="hrefs">
    <uri val="/car/moved" />
  </list>
</obj>

<obj is="obix:WatchOut">
  <list names="values">
    <feed href="/car/moved" of="/def/Coordinate/" />  <!-- empty history -->
  </list>
</obj>
```

Now every time we call `pollChanges` for the Watch, the server will send us the list of event instances which have accumulated since our last poll:

```
<obj is="obix:WatchOut">
  <list names="values">
    <feed href="/car/moved" of="/def/Coordinate">
      <obj>
        <real name="lat"  val="37.645022"/>
        <real name="long" val="-77.575851"/>
      </obj>
      <obj>
        <real name="lat"  val="37.639046"/>
        <real name="long" val="-77.61872"/>
      </obj>
    </feed>
  </list>
</obj>
```

Note the feed's `of` attribute works just like the `list`'s `of` attribute. The children event instances are assumed to inherit the Contract defined by `of` unless explicitly overridden. If an event instance does override the `of` Contract, then it MUST be Contract compatible. Refer to the rules defined in Section 7.8.

Invoking a `pollRefresh` operation on a Watch with a feed that has an event history, SHOULD return all the historical events as if the `pollRefresh` was an `add` operation. If an event history is not available, then `pollRefresh` SHOULD act like a normal `pollChanges` and just return the events which have occurred since the last poll.

13 Points

Anyone familiar with automation systems immediately identifies with the term *Point* (sometimes called *tags* in the industrial space). Although there are many different definitions, generally points map directly to a sensor or actuator (called *Hard Points*). Sometimes the concept of a Point is mapped to a configuration variable such as a software setpoint (called *Soft Points*). In some systems Point is an atomic value, and in others it encapsulates a great deal of status and configuration information.

The goal of OBIX is to capture a normalization representation of Points without forcing an impedance mismatch on implementers trying to make their native system OBIX accessible. To meet this requirement, OBIX defines a low level abstraction for Point - simply one of the primitive value types with associated status information. Point is basically just a marker Contract used to tag an Object as exhibiting “Point” semantics:

```
<obj href="obix:Point"/>
```

This Contract MUST only be used with the value primitive types: `bool`, `real`, `enum`, `str`, `abstime`, and `reltime`. Points SHOULD use the `status` attribute to convey quality information. This Table specifies how to map common control system semantics to a value type:

Point type	OBIX Object	Example
digital Point	`bool`	`<bool is="obix:Point" val="true"/>`
analog Point	`real`	`<real is="obix:Point" val="22" unit="obix:units/celsius"/>`
multi-state Point	`enum`	`<enum is="obix:Point" val="slow"/>`

Table 13-1. Base Point types.

13.1 Writable Points

Different control systems handle Point writes using a wide variety of semantics. Sometimes we write a Point at a specific priority level. Sometimes we override a Point for a limited period of time, after which the Point falls back to a default value. The OBIX specification does not attempt to impose a specific model on implementers. Rather OBIX provides a standard `WritablePoint` Contract which may be extended with additional mixins to handle special cases. `WritablePoint` defines write as an operation which takes a `WritePointIn` structure containing the value to write. The Contracts are:

```
<obj href="obix:WritablePoint" is="obix:Point">
  <op name="writePoint" in="obix:WritePointIn" out="obix:Point"/>
</obj>

<obj href="obix:WritePointIn">
  <obj name="value"/>
</obj>
```

It is implied that the value passed to `writePoint` MUST match the type of the Point. For example if `WritablePoint` is used with an `enum`, then `writePoint` MUST pass an `enum` for the value.

14 History

Most automation systems have the ability to persist periodic samples of point data to create a historical archive of a point’s value over time. This feature goes by many names including logs, trends, or histories. In OBIX, a *history* is defined as a list of time stamped point values. The following features are provided by OBIX histories:

History Object	A normalized representation for a history itself
History Record	A record of a point sampling at a specific timestamp
History Query	A standard way to query history data as Points
History Rollup	A standard mechanism to do basic rollups of history data
History Append	The ability to push new history records into a history

Table 14-1. Features of OBIX Histories.

14.1 History Object

Any Object which wishes to expose itself as a standard OBIX history implements the `obix:History` Contract:

```
<obj href="obix:History">
  <int     name="count"  min="0" val="0"/>
  <abstime name="start"  null="true"/>
  <abstime name="end"    null="true"/>
  <str     name="tz"     null="true"/>
  <list    name=”formats” of=”obix:str” null=”true”/>
  <op      name="query"  in="obix:HistoryFilter" out="obix:HistoryQueryOut"/>
  <feed    name="feed"   in="obix:HistoryFilter" of="obix:HistoryRecord"/>
  <op      name="rollup" in="obix:HistoryRollupIn" out="obix:HistoryRollupOut"/>
  <op      name="append" in="obix:HistoryAppendIn" out="obix:HistoryAppendOut"/>
</obj>
```

The child properties of `obix:History` are:

Property	Description
`count`	The number of history records contained by the history
`start`	Provides the timestamp of the oldest record. The timezone of this abstime MUST match `History.tz`
`end`	Provides the timestamp of the newest record. The timezone of this abstime MUST match `History.tz`
`tz`	A standardized timezone identifier for the history data (see Section 4.1.11)
`formats`	Provides a list of strings describing the formats in which the server can provide the history data
`query`	The operation used to query the history to read history records
`feed`	The object used to subscribe to a real-time feed of history records
`rollup`	The operation used to perform history rollups (it is only supported for numeric history data)

append	The operation used to push new history records into the history

Table 14-2. Properties of obix:History.

An example of a history which contains an hour of 15 minute temperature data:

```
<obj href="http://x/outsideAirTemp/history/" is="obix:History">
  <int     name="count"  val="5"/>
  <abstime name="start"  val="2005-03-16T14:00:00-05:00" tz="America/New_York"/>
  <abstime name="end"    val="2005-03-16T15:00:00-05:00" tz="America/New_York"/>
  <str     name="tz"     val="America/New_York"/>
  <list    name="formats" of="obix:str">
    <str val="text/csv"/>
  </list>
  <op      name="query"  href="query"/>
  <op      name="rollup" href="rollup"/>
</obj>
```

14.2 History Queries

Every History Object contains a query operation to query the historical data. A client MAY invoke the query operation to request the data from the server as an obix:HistoryQueryOut. Alternatively, if the server is able to provide the data in a different format, such as CSV, it SHOULD list these additionally supported formats in the formats field. A client MAY then supply one of these defined formats in the HistoryFilter input query.

14.2.1 HistoryFilter

The History.query input Contract:

```
<obj href="obix:HistoryFilter">
  <int     name="limit"    null="true"/>
  <abstime name="start"    null="true"/>
  <abstime name="end"      null="true"/>
  <str     name="format"   null="true"/>
  <bool    name="compact" val="false"/>
</obj>
```

These fields are described in detail in this Table:

Field	Description
limit	An integer indicating the maximum number of records to return. Clients can use this field to throttle the amount of data returned by making it non-null. Servers MUST never return more records than the specified limit. However servers are free to return fewer records than the limit.
start	If non-null this field indicates an inclusive lower bound for the query's time range. This value SHOULD match the history's timezone, otherwise the server MUST normalize based on absolute time.
end	If non-null this field indicates an inclusive upper bound for the query's time range. This value SHOULD match the history's timezone, otherwise the server MUST normalize based on absolute time.
format	If non-null this field indicates the format that the client is requesting for the returned data. If the client uses this field the server MUST return a HistoryQueryOut with a non-null dataRef URI, or return an error if it is unable to supply the requested format. A client SHOULD use one of the formats defined in the History's formats field when using this field in the filter.
compact	If non-null and true, this field indicates the client is requesting the data in the compact format described below. If false or null, the server MUST return the data in the standard format compatible with the 1.0 specification.

Table 14-3. Properties of `obix:HistoryFilter`.

14.2.2 HistoryQueryOut

The `History.query` output Contract:

```
<obj href="obix:HistoryQueryOut">
  <int     name="count" min="0" val="0"/>
  <abstime name="start" null="true"/>
  <abstime name="end"   null="true"/>
  <list    name="data"  of="obix:HistoryRecord" null="true"/>
  <uri     name="dataRef" null="true"/>
</obj>
```

Just like `History`, every `HistoryQueryOut` returns `count`, `start`, and `end`. But unlike `History`, these values are for the query result, not the entire history. The actual history data is stored as a list of `HistoryRecords` in the `data` field. Remember that child order is not guaranteed in OBIX, therefore it might be common to have `count` after `data`. The start, end, and data HistoryRecord timestamps MUST have a timezone which matches `History.tz`.

When using a client-requested format, the server MUST provide a URI that can be followed by the client to obtain the history data in the alternate format. The exact definition of this format is out of scope of this specification, but SHOULD be agreed upon by both the client and server.

14.2.3 HistoryRecord

The `HistoryRecord` Contract specifies a record in a history query result:

```
<obj href="obix:HistoryRecord">
  <abstime name="timestamp" null="true"/>
  <obj     name="value"     null="true"/>
</obj>
```

Typically the value SHOULD be one of the value types used with `obix:Point`.

14.2.4 History Query Examples

Let’s examine an example query from the “/outsideAirTemp/history” example above.

14.2.4.1 History Query as OBIX Objects

First let’s see how a client and server interact using the standard history query mechanism:

Client invoke request:

```
INVOKE http://x/outsideAirTemp/history/query
<obj name="in" is="obix:HistoryFilter">
  <int     name="limit" val="5"/>
  <abstime name="start" val="2005-03-16T14:00:00-05:00" tz="America/New_York"/>
</obj>
```

Server response:

```
<obj href="http://x/outsideAirTemp/history/query" is="obix:HistoryQueryOut">
  <int name="count" val="5"/>
  <abstime name="start" val="2005-03-16T14:00:00-05:00" tz="America/New_York"/>
  <abstime name="end"   val="2005-03-16T15:00:00-05:00" tz="America/New_York"/>
  <reltime name="interval" val="PT15M"/>
  <list name="data" of="#RecordDef obix:HistoryRecord">
    <obj> <abstime name="timestamp" val="2005-03-16T14:00:00-05:00"/>
          <real name="value" val="40"/> </obj>
    <obj> <abstime name="timestamp"  val="2005-03-16T14:15:00-05:00"/>
          <real name="value" val="42"/> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T14:30:00-05:00"/>
          <real name="value" val="43"/> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T14:45:00-05:00"/>
          <real name="value" val="47"/> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T15:00:00-05:00"/>
          <real name="value" val="44"/> </obj>
  </list>
```



```
  <obj href="#RecordDef" is="obix:HistoryRecord">
    <abstime name="timestamp" tz="America/New_York"/>
    <real name="value" unit="obix:units/fahrenheit"/>
  </obj>
</obj>
```

Note in the example above how the `data` list uses a document local Contract to define Facets common to all the records (although we still have to flatten the Contract List).

14.2.4.2 History Query as Preformatted List

Now let’s see how this might be done in a more compact format. The server in this case is able to return the history data as a CSV list.

Client invoke request:

```
INVOKE http://myServer/obix/outsideAirTemp/history/query
<obj name=”in” is=”obix:HistoryFilter”>
  <int     name="limit" val=”5”/>
  <abstime name="start" val=”2005-03-16T14:00:00-05:00” tz="America/New_York"/>
  <str name=”format” val=”text/csv”/>
</obj>
```

Server response:

```
<obj href="http://myServer/obix/outsideAirTemp/history/query" is="obix:HistoryQueryOut">
  <int name="count" val="5”/>
  <abstime name="start" val="2005-03-16T14:00:00-05:00" tz="America/New_York"/>
  <abstime name="end"   val="2005-03-16T15:00:00-05:00" tz="America/New_York"/>
  <uri name=”dataRef” val=”http://x/outsideAirTemp/history/query?text/csv”/>
</obj>
```

Client then reads the dataRef URI:

```
GET http://x/outsideAirTemp/history/query?text/csv
```

Server response:

```
2005-03-16T14:00:00-05:00,40
2005-03-16T14:15:00-05:00,42
2005-03-16T14:30:00-05:00,43
2005-03-16T14:45:00-05:00,47
2005-03-16T15:00:00-05:00,44
```

Note that the client’s second request is NOT an OBIX request, and the subsequent server response is NOT an OBIX document, but just arbitrarily formatted data as requested by the client – in this case text/csv. Also it is important to note that this is simply an example. While the usage of the format and dataRef properties is normative, the usage of the text/csv MIME type and how the data is actually presented is purely non-normative. It is not intended to suggest CSV as a mechanism for how the data should be formatted, as that is an agreement to be made between the client and server. The server and client are free to use any agreed-upon format, for example, one where the timestamps are inferred rather than repeated, for maximum brevity.

14.2.5 Compact Histories

When a server contains a large number of history records, it is important to be as concise as possible when retrieving the records. The `HistoryRecord` format is fine for small histories, but it is not uncommon for servers to contain thousands, or tens of thousands, of data points, or even more. To allow a more concise representation of the historical data, a client MAY request that the server provide the query output in a “compact” format. This is done by setting the `compact` attribute of the HistoryFilter Contract to true. The server MUST then respond with a `CompactHistoryQueryOut` if it supports compact history reporting for the referenced History, or an error if it does not.

The `CompactHistoryQueryOut` Contract is:

```
<obj href=”obix:CompactHistoryQueryOut” is=”obix:HistoryQueryOut”>
  <reltime name=”interval” null=”true”/>
  <str     name=”delimiter”/>
```



```
  <list    name="data" of="obix:CompactHistoryRecord" null="true"/>
</obj>
```

Note that the data element is narrowed to require the `CompactHistoryRecord` type, which is defined as:

```
<str href="obix:CompactHistoryRecord" is="obix:HistoryRecord"/>
```

The `CompactHistoryRecord` Contract narrows the `HistoryRecord` Contract to the `str` element type. The semantic requirements of the Contract allow for a more compact representation of the record as an OBIX Object, although with some restrictions:

- The `timestamp` and `value` child elements MUST be null when encoded. These are determined from the `val` attribute.
- The `val` attribute of the `CompactHistoryRecord` MUST be a string containing a delimited list of entities matching the record definition. The delimiter MUST be included using the `delimiter` element of the `CompactHistoryQueryOut`.
- The record definition MUST be provided in an accessible URI to the client. The record definition SHOULD be provided in a document-local Contract defining the type of each item in the record, as well as any Facets that apply to every record's fields.
- The `CompactHistoryRecord` MUST be interpreted by inserting each item in the delimited list contained in the `val` attribute into the respective child element's `val` attribute.
- For histories with regular collection intervals, the `timestamp` field MAY be left empty, if it can be inferred by the consumer. If the `timestamp` field is left empty on any record, the server MUST include the `interval` element in the `HistoryQueryOut`. Consumers MUST be able to handle existence or non-existence of the `timestamp` field. Note that this only applies when the timestamp matches the expected value based on the collection interval of the history. If a record exists at an irregular time interval, such as for skipped records or COV histories, the timestamp MUST be included in the record.
- The interpretation of the `CompactHistoryRecord` MUST be identical to the interpretation of a `HistoryRecord` with the same list of values described as child elements.
- A consumer of the `CompactHistoryRecord` MAY skip the actual internal conversion of the `CompactHistoryRecord` into its expanded form, and use a 'smart' decoding process to consume the list as if it were presented in the `HistoryRecord` form.

14.2.5.1 CompactHistoryRecord Example

Let's look at the same scenario as in our previous example, this time expressed using `CompactHistoryRecords`. The server is providing additional information with certain elements; this is reflected in the record definition at the end.

Client invoke request:

```
INVOKE http://x/outsideAirTemp/history/query
<obj name="in" is="obix:HistoryFilter">
  <int     name="limit" val="5"/>
  <abstime name="start" val="2005-03-16T14:00:00-05:00" tz="America/New_York"/>
  <bool    name="compact" val="true"/>
</obj>
```

Server response:

```
<obj href="http://x/outsideAirTemp/history/query" is="obix:CompactHistoryQueryOut">
  <int name="count" val="5"/>
  <abstime name="start" val="2005-03-16T14:00:00-05:00" tz="America/New_York"/>
  <abstime name="end"   val="2005-03-16T15:00:00-05:00" tz="America/New_York"/>
  <reltime name="interval" val="PT15M"/>
  <str name="delimiter" val=","/>
  <list name="data" of="#RecordDef obix:CompactHistoryRecord">
    <str val=",40,44"/>  <!-- may be inferred from start -->
    <str val=",42,45"/>  <!-- regular collection, inferred -->
    <str val="2005-03-16T14:30:02-05:00,43,48"/>  <!-- irregular timestamp -->
    <str val=",47,"/>  <!-- inferred, dischgTemp not available -->
    <str val=",44,47"/>  <!-- inferred -->
  </list>
```



```
  <obj href="#RecordDef" is="obix:CompactHistoryRecord">
    <abstime name="timestamp" tz="America/New_York"/>
    <real name="value" unit="obix:units/fahrenheit"/>
    <real name=”dischargeAirTemp” unit=”obix:units/fahrenheit”/>
  </obj>
```

14.3 History Rollups

Control systems collect historical data as raw time sampled values. However, most applications wish to consume historical data in a summarized form which we call *rollups*. The rollup operation is used to summarize an interval of time. History rollups only apply to histories which store numeric information. Attempting to query a rollup on a non-numeric history SHOULD result in an error.

14.3.1 HistoryRollupIn

The `History.rollup` input Contract extends `HistoryFilter` to add an interval parameter:

```
<obj href="obix:HistoryRollupIn" is="obix:HistoryFilter">
  <reltime name="interval"/>
</obj>
```

14.3.2 HistoryRollupOut

The `History.rollup` output Contract:

```
<obj href="obix:HistoryRollupOut">
  <int     name="count" min="0" val="0"/>
  <abstime name="start" null="true"/>
  <abstime name="end"   null="true"/>
  <list name="data" of="obix:HistoryRollupRecord"/>
</obj>
```

The `HistoryRollupOut` Object looks very much like `HistoryQueryOut` except it returns a list of `HistoryRollupRecords`, rather than `HistoryRecords`. Note: unlike `HistoryQueryOut`, the `start` for `HistoryRollupOut` is exclusive, not inclusive. This issue is discussed in greater detail next. The start, end, and data `HistoryRollupRecord` timestamps MUST have a timezone which matches `History.tz`.

14.3.3 HistoryRollupRecord

A history rollup returns a list of `HistoryRollupRecords`:

```
<obj href="obix:HistoryRollupRecord">
  <abstime name="start"/>
  <abstime name="end"  />
  <int  name="count"/>
  <real name="min" />
  <real name="max" />
  <real name="avg" />
  <real name="sum" />
</obj>
```

The children are defined in the Table below:

Property	Description
`start`	The exclusive start time of the record’s rollup interval
`end`	The inclusive end time of the record’s rollup interval
`count`	The number of records used to compute this rollup interval
`min`	The minimum value of all the records within the interval
`max`	The maximum value of all the records within the interval

avg	The arithmetic mean of all the values within the interval
sum	The summation of all the values within the interval

Table 14-4. Properties of `obix:HistoryRollupRecord`.

14.3.4 Rollup Calculation

The best way to understand how rollup calculations work is through an example. Let’s consider a history of meter data where we collected two hours of 15 minute readings of kilowatt values:

```
<obj is="obix:HistoryQueryOut">
  <int     name="count" val="9">
  <abstime name="start" val="2005-03-16T12:00:00+04:00" tz="Asia/Dubai"/>
  <abstime name="end"   val="2005-03-16T14:00:00+04:00" tz="Asia/Dubai"/>
  <list name="data" of="#HistoryDef obix:HistoryRecord">
    <obj> <abstime name="timestamp" val="2005-03-16T12:00:00+04:00"/>
          <real name="value" val="80"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T12:15:00+04:00"/>
          <real name="value" val="82"></obj>
    <obj> <abstime name="timestamp" val="2005-03-16T12:30:00+04:00"/>
          <real name="value" val="90"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T12:45:00+04:00"/>
          <real name="value" val="85"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T13:00:00+04:00"/>
          <real name="value" val="81"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T13:15:00+04:00"/>
          <real name="value" val="84"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T13:30:00+04:00"/>
          <real name="value" val="91"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T13:45:00+04:00"/>
          <real name="value" val="83"> </obj>
    <obj> <abstime name="timestamp" val="2005-03-16T14:00:00+04:00"/>
          <real name="value" val="78"> </obj>
  </list>
  <obj href="#HistoryRecord" is="obix:HistoryRecord">
    <abstime name="timestamp" tz="Asia/Dubai"/>
    <real name="value" unit="obix:units/kilowatt"/>
  <obj>
</obj>
```

If we were to query the rollup using an interval of 1 hour with a start time of 12:00 and end time of 14:00, the result should be:

```
<obj is="obix:HistoryRollupOut obix:HistoryQueryOut">
  <int     name="count" val="2">
  <abstime name="start" val="2005-03-16T12:00:00+04:00 tz="Asia/Dubai"/>
  <abstime name="end"   val="2005-03-16T14:00:00+04:00" tz="Asia/Dubai"/>
  <list name="data" of="obix:HistoryRollupRecord">
    <obj>
      <abstime name="start" val="2005-03-16T12:00:00+04:00"
               tz="Asia/Dubai"/>
      <abstime name="end"   val="2005-03-16T13:00:00+04:00"
               tz="Asia/Dubai"/>
      <int  name="count" val="4"    />
      <real name="min"   val="81"   />
      <real name="max"   val="90"   />
      <real name="avg"   val="84.5" />
      <real name="sum"   val="338"  />
    </obj>
    <obj>
      <abstime name="start" val="2005-03-16T13:00:00+04:00"
               tz="Asia/Dubai"/>
      <abstime name="end"   val="2005-03-16T14:00:00+04:00"
               tz="Asia/Dubai"/>
      <int  name="count" val="4"   />
      <real name="min"   val="78"  />
      <real name="max"   val="91"  />
      <real name="avg"   val="84"  />
      <real name="sum"   val="336" />
    </obj>
```



```
  </list>
</obj>
```

The first item to notice is that the first raw record of 80kW was never used in the rollup. This is because start time is always exclusive. The reason start time has to be exclusive is because we are summarizing discrete samples into a contiguous time range. It would be incorrect to include a record in two different rollup intervals! To avoid this problem we always make start time exclusive and end time inclusive. The following Table illustrates how the raw records were applied to rollup intervals:

Interval Start (exclusive)	Interval End (inclusive)	Records Included
2005-03-16T12:00	2005-03-16T13:00	82 + 90 + 85 + 81 = 338
2005-03-16T13:00	2005-03-16T14:00	84 + 91 + 83 + 78 = 336

Table 14-5. Calculation of OBIX History rollup values.

14.4 History Feeds

The `History` Contract specifies a feed for subscribing to a real-time feed of the history records. `History.feed` reuses the same `HistoryFilter` input Contract used by `History.query` – the same semantics apply. When adding a History feed to a Watch, the initial result SHOULD contain the list of `HistoryRecords` filtered by the input parameter (the initial result should match what `History.query` would return). Subsequent calls to `Watch.pollChanges` SHOULD return any new `HistoryRecords` which have been collected since the last poll that also satisfy the `HistoryFilter`.

14.5 History Append

The `History.append` operation allows a client to push new `HistoryRecords` into a History log (assuming proper security credentials). This operation comes in handy when bi-direction HTTP connectivity is not available. For example if a device in the field is behind a firewall, it can still push history data on an interval basis to a server using the append operation.

14.5.1 HistoryAppendIn

The `History.append` input Contract:

```
<obj href="obix:HistoryAppendIn">
  <list name="data" of="obix:HistoryRecord"/>
</obj>
```

The `HistoryAppendIn` is a wrapper for the list of `HistoryRecords` to be inserted into the History. The `HistoryRecords` SHOULD use a timestamp which matches `History.tz`. If the timezone doesn't match, then the server MUST normalize to its configured timezone based on absolute time. The `HistoryRecords` in the data list MUST be sorted by timestamp from oldest to newest, and MUST not include a timestamp equal to or older than `History.end`.

14.5.2 HistoryAppendOut

The `History.append` output Contract:

```
<obj href="obix:HistoryAppendOut">
  <int     name="numAdded"/>
  <int     name="newCount"/>
  <abstime name="newStart" null="true"/>
  <abstime name="newEnd"   null="true"/>
</obj>
```

The output of the append operation returns the number of new records appended to the History and the new total count, start time, and end time of the entire History. The newStart and newEnd timestamps MUST have a timezone which matches `History.tz`.

15 Alarming

OBIX specifies a normalized model to query, Watch, and acknowledge alarms. In OBIX, an alarm indicates a condition which requires notification of either a user or another application. In many cases an alarm requires acknowledgement, indicating that someone (or something) has taken action to resolve the alarm condition. The typical lifecycle of an alarm is:

1. **Source Monitoring**: algorithms in a server monitor an *alarm source*. An alarm source is an Object with an href which has the potential to generate an alarm. Example of alarm sources might include sensor points (this room is too hot), hardware problems (disk is full), or applications (building is consuming too much energy at current energy rates)
2. **Alarm Generation**: if the algorithms in the server detect that an alarm source has entered an alarm condition, then an *alarm* record is generated. Every alarm is uniquely identified using an href and represented using the `obix:Alarm` Contract. Sometimes we refer to the alarm transition as *off-normal*.
3. **To Normal**: many alarm sources are said to be *stateful* - eventually the alarm source exits the alarm state, and is said to return *to-normal*. Stateful alarms implement the `obix:StatefulAlarm` Contract. When the source transitions to normal, we update `normalTimestamp` of the alarm.
4. **Acknowledgement**: often we require that a user or application acknowledges that they have processed an alarm. These alarms implement the `obix:AckAlarm` Contract. When the alarm is acknowledged, we update `ackTimestamp` and `ackUser`.

15.1 Alarm States

Alarm state is summarized with two variables:

In Alarm	Is the alarm source currently in the alarm condition or in the normal condition? This variable maps to the `alarm` status state.
Acknowledged	Is the alarm acknowledged or unacknowledged? This variable maps to the `unacked` status state.

Table 15-1. Alarm states in OBIX.

Either of these states may transition independent of the other. For example an alarm source can return to normal before or after an alarm has been acknowledged. Furthermore it is not uncommon to transition between normal and off-normal multiple times generating several alarm records before any acknowledgements occur.

Note not all alarms have state. An alarm which implements neither `StatefulAlarm` nor the `AckAlarm` Contracts is completely stateless – these alarms merely represent event. An alarm which implements `StatefulAlarm` but not `AckAlarm` will have an in-alarm state, but not acknowledgement state. Conversely an alarm which implements `AckAlarm` but not `StatefulAlarm` will have an acknowledgement state, but not in-alarm state.

15.1.1 Alarm Source

The current alarm state of an alarm source is represented using the `status` attribute. This attribute is discussed in Section 4.1.10. It is recommended that alarm sources always report their status via the `status` attribute.

15.1.2 StatefulAlarm and AckAlarm

An `Alarm` record is used to summarize the entire lifecycle of an alarm event. If the alarm implements `StatefulAlarm` it tracks transition from off-normal back to normal. If the alarm implements `AckAlarm`, then it also summarizes the acknowledgement. This allows for four discrete alarm states, which are described in terms of the alarm Contract properties:

Alarm State	alarm	acked	normalTimestamp	ackTimestamp
new unacked alarm	true	false	null	null
acknowledged alarm	true	true	null	non-null
unacked returned alarm	false	false	non-null	null
acked returned alarm	false	true	non-null	non-null

Table 15-2. Alarm lifecycle states in OBIX.

15.2 Alarm Contracts

15.2.1 Alarm

The core `Alarm` Contract is:

```
<obj href="obix:Alarm">
  <ref name="source"/>
  <abstime name="timestamp"/>
</obj>
```

The child Objects are:

- **source**: the URI which identifies the alarm source. The source SHOULD reference an OBIX Object which models the entity that generated the alarm.
- **timestamp**: this is the time at which the alarm source transitioned from normal to off-normal and the Alarm record was created.

15.2.2 StatefulAlarm

Alarms which represent an alarm state which may transition back to normal SHOULD implement the `StatefulAlarm` Contract:

```
<obj href="obix:StatefulAlarm" is="obix:Alarm">
  <abstime name="normalTimestamp" null="true"/>
</obj>
```

The child Object is:

- **normalTimestamp**: if the alarm source is still in the alarm condition, then this field is null. Otherwise this indicates the time of the transition back to the normal condition.

15.2.3 AckAlarm

Alarms which support acknowledgment SHOULD implement the `AckAlarm` Contract:

```
<obj href="obix:AckAlarm" is="obix:Alarm">
  <abstime name="ackTimestamp" null="true"/>
  <str name="ackUser" null="true"/>
  <op name="ack" in="obix:AckAlarmIn" out="obix:AckAlarmOut"/>
</obj>

<obj href="obix:AckAlarmIn">
  <str name="ackUser" null="true"/>
</obj>

<obj href="obix:AckAlarmOut">
```



```
  <obj name="alarm" is="obix:AckAlarm obix:Alarm"/>
</obj>
```

The child Objects are:

- **ackTimestamp**: if the alarm is unacknowledged, then this field is null. Otherwise this indicates the time of the acknowledgement.
- **ackUser**: if the alarm is unacknowledged, then this field is null. Otherwise this field should provide a string indicating who was responsible for the acknowledgement.

The `ack` operation is used to programmatically acknowledge the alarm. The client may optionally specify an `ackUser` string via Ack`AlarmIn`. However, the server is free to ignore this field depending on security conditions. For example a highly trusted client may be allowed to specify its own `ackUser`, but a less trustworthy client may have its `ackUser` predefined based on the authentication credentials of the protocol binding. The `ack` operation returns an `AckAlarmOut` which contains the updated alarm record. Use the `Lobby.batch` operation to efficiently acknowledge a set of alarms.

15.2.4 PointAlarms

It is very common for an alarm source to be an `obix:Point`. A respective `PointAlarm` Contract is provided as a normalized way to report the value which caused the alarm condition:

```
<obj href="obix:PointAlarm" is="obix:Alarm">
  <obj name="alarmValue"/>
</obj>
```

The `alarmValue` Object SHOULD be one of the value types defined for `obix:Point` in Section 13.

15.3 AlarmSubject

Servers which implement OBIX alarming MUST provide one or more Objects which implement the `AlarmSubject` Contract. The `AlarmSubject` Contract provides the ability to categorize and group the sets of alarms a client may discover, query, and watch. For instance a server could provide one `AlarmSubject` for all alarms and other `AlarmSubjects` based on priority or time of day. The Contract for `AlarmSubject` is:

```
<obj href="obix:AlarmSubject">
  <int     name="count"  min="0" val="0"/>
  <op      name="query" in="obix:AlarmFilter" out="obix:AlarmQueryOut"/>
  <feed    name="feed"  in="obix:AlarmFilter" of="obix:Alarm"/>
</obj>

<obj href="obix:AlarmFilter">
  <int     name="limit"  null="true"/>
  <abstime name="start"  null="true"/>
  <abstime name="end"    null="true"/>
</obj>

<obj href="obix:AlarmQueryOut">
  <int     name="count" min="0" val="0"/>
  <abstime name="start" null="true"/>
  <abstime name="end"   null="true"/>
  <list    name="data"  of="obix:Alarm"/>
</obj>
```

The `AlarmSubject` follows the same design pattern as `History`. The `AlarmSubject` specifies the active `count` of alarms; however, unlike `History` it does not provide the `start` and `end` bounding timestamps. It contains a `query` operation to read the current list of alarms with an `AlarmFilter` to filter by time bounds. `AlarmSubject` also contains a feed Object which may be used to subscribe to the alarm events.

15.4 Alarm Feed Example

The following example illustrates how a feed works with this `AlarmSubject`:

```
<obj is="obix:AlarmSubject" href="/alarms/">
  <int  name="count" val="2"/>
```



```
  <op   name="query" href="query"/>
  <feed name="feed"  href="feed" />
</obj>
The server indicates it has two open alarms under the specified AlarmSubject. If a client
were to add the AlarmSubject’s feed to a watch:
<obj is="obix:WatchIn">
 <list names="hrefs"/>
  <uri val="/alarms/feed">
    <obj name="in" is="obix:AlarmFilter">
      <int name="limit" val="25"/>
    </obj>
  </uri>
 </list>
</obj>

<obj is="obix:WatchOut">
 <list names="values">
   <feed href="/alarms/feed" of="obix:Alarm">
    <obj href="/alarmdb/528" is="obix:StatefulAlarm obix:PointAlarm obix:Alarm">
      <ref name="source" href="/airHandlers/2/returnTemp"/>
      <abstime name="timestamp" val="2006-05-18T14:20:00Z"/>
      <abstime name="normalTimestamp"  null="true"/>
      <real name="alarmValue" val="80.2"/>
    </obj>
    <obj href="/alarmdb/527" is="obix:StatefulAlarm obix:PointAlarm obix:Alarm">
      <ref name="source" href="/doors/frontDoor"/>
      <abstime name="timestamp" val="2006-05-18T14:18:00Z"/>
      <abstime name=" normalTimestamp" null="true"/>
      <real name="alarmValue" val="true"/>
    </obj>
  </feed>
 </list>
</obj>
```

The Watch returns the historic list of alarm events which is two open alarms. The first alarm indicates an out of bounds condition in AirHandler-2’s return temperature. The second alarm indicates that the system has detected that the front door has been propped open.

Now let’s fictionalize that the system detects the front door is closed, and alarm point transitions to the normal state. The next time the client polls the Watch the alarm would show up in the feed list (along with any additional changes or new alarms not shown here):

```
<obj is="obix:WatchOut">
 <list names="values">
   <feed href="/alarms/feed" of="obix:Alarm">>
    <obj href="/alarmdb/527" is="obix:StatefulAlarm obix:PointAlarm obix:Alarm">
      <ref name="source" href="/doors/frontDoor"/>
      <abstime name="timestamp" val="2006-05-18T14:18:00Z"/>
      <abstime name=" normalTimestamp" val="2006-05-18T14:45:00Z"/>
      <real name="alarmValue" val="true"/>
    </obj>
   </feed>
 </list>
</obj>
```


16 Security

Security is a broad topic that covers many issues. Some of the main concepts are listed below:

Authentication	Verifying a user (client) is who they claim to be
Encryption	Protecting OBIX documents from viewing by unauthorized entities
Permissions	Checking a user's permissions before granting access to read/write Objects or invoke operations
User Management	Managing user accounts and permissions levels

Table 16-1. Security concepts for OBIX.

The basic philosophy of OBIX is to leave these issues outside of the specification. Authentication and encryption are left as a protocol binding issue. Privileges and user management are left as a vendor implementation issue. Although it is entirely possible to define a publicly exposed user management model through OBIX, this specification does not define any standard Contracts for user management.

16.1 Error Handling

It is expected that an OBIX server will perform authentication and utilize those user credentials for checking permissions before processing read, write, and invoke requests. As a general rule, servers SHOULD return `err` with the `obix:PermissionErr` Contract to indicate a client lacks the permission to perform a request. In particularly sensitive applications, a server may instead choose to return `BadUriErr` so that an untrustworthy client is unaware that a specific object even exists.

16.2 Permission-based Degradation

Servers SHOULD strive to present their object model to a client based on the privileges available to the client. This behavior is called *permission based degradation*. The following rules summarize effective permission based degradation:

1. If an Object cannot be read, then it SHOULD NOT be discoverable through Objects which are available.
2. Servers SHOULD attempt to group standard Contracts within the same privilege level – for example don't split `obix:History`'s `start` and `end` into two different security levels such that a client might be able to read `start`, and not `end`.
3. Servers SHOULD NOT include a Contract in an Object's `is` attribute if the Contract's children are not readable to the client.
4. If an Object isn't writable, then the `writable` attribute SHOULD be set to `false` (either explicitly or through a Contract default).
5. If an `op` inherited from a visible Contract cannot be invoked, then the server SHOULD set the `null` attribute to `true` to disable it.

17 Conformance

An implementation is conformant with this specification if it satisfies all of the MUST and REQUIRED level requirements defined herein for the functions implemented. Normative text within this specification takes precedence over normative outlines, which in turn take precedence over the **XML Schema** and **WSDL** descriptions, which in turn take precedence over examples.

An implementation is a conforming OBIX Server if it meets the conditions described in Section 17.1. An implementation is a conforming OBIX Client if it meets the conditions described in Section 17.2. An implementation is a conforming OBIX Server and a conforming OBIX Client if it meets the conditions of both Section 17.1 and Section 17.2.

17.1 Conditions for a Conforming OBIX Server

An implementation conforms to this specification as an OBIX Server if it meets the conditions described in the following subsections. OBIX servers MUST implement the OBIX Lobby Object.

17.1.1 Lobby

A conforming OBIX server MUST meet the following conditions to satisfy the Lobby Conformance Clause:

1. OBIX Servers MUST have an accessible Object which implements the `obix:Lobby` Contract.
2. The Lobby MUST provide a `<ref>` to an Object which implements the `obix:About` Contract.
3. The Lobby MUST provide a `<ref>` to an Object which implements the `obix:WatchService` Contract.
4. The Lobby MUST provide an `<op>` to invoke batch operations using the `obix:BatchIn` and `obix:BatchOut` Contracts.
5. The Lobby MUST provide a list of the encodings supported.
6. The Lobby MUST provide a list of the bindings supported.

17.1.2 Bindings

An implementation MUST support one of the bindings defined in the companion specifications to this specification that describe OBIX Bindings.

17.1.3 Encodings

An implementation MUST support one of the encodings defined in the companion specification to this specification, **OBIX Encodings**. An implementation SHOULD support the XML encoding, as this encoding is used by the majority of OBIX implementations. An implementation MUST support negotiation of the encoding to be used with a client according to the mechanism defined for the specific binding used.

An implementation MUST return values according to the rules defined in Section 4. For example, an implementation MUST encode `bool` Objects' `val` attribute using the literals "true" and "false" only.

17.1.4 Contracts

An implementation MUST flatten Contract hierarchies when reporting them in an OBIX document, according to Section 7.6.1.

17.2 Conditions for a Conforming OBIX Client

An implementation conforms to this specification as an OBIX Client if it meets the conditions described in the following subsections.

17.2.1 Encoding

An implementation MUST support one of the encodings defined in this specification. An implementation SHOULD support the XML encoding, as this encoding is used by the majority of OBIX implementations. An implementation MUST support negotiation of which encoding to use in communicating with an OBIX server using the mechanism defined for the binding being used.

17.2.2 Naming

An implementation MUST be able to interpret and navigate URI schemes according to the general rules described in section 6.3. An implementation SHOULD be able to interpret and navigate HTTP URIs, as this is used by the majority of OBIX Server implementations.

17.2.3 Contracts

An implementation MUST be able to consume and use OBIX Contracts defined by OBIX Server implementations with which it interacts.

Appendix A. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

Ron Ambrosio, IBM
Brad Benson, Trane
Ron Bernstein, LonMark International*
Ludo Bertsch, Continental Automated Buildings Association (CABA)
Chris Bogen, US Department of Defense
Rich Blomseth, Echelon Corporation
Anto Budiardjo, Clasma Events, Inc.
Jochen Burkhardt, IBM
JungIn Choi, Kyungwon University
David Clute, Cisco Systems, Inc.*
Toby Considine, University of North Carolina at Chapel Hill
William Cox, Individual
Robert Dolin, Echelon Corporation
Marek Dziedzic, Treasury Board of Canada, Secretariat
Brian Frank, SkyFoundry
Craig Gemmill, Tridium, Inc.
Matthew Giannini, Tridium, Inc.
Markus Jung, Vienna University of Technology
Christopher Kelly, Cisco Systems
Wonsuk Ko, Kyungwon University
Perry Krol, TIBCO Software Inc.
Corey Leong, Individual
Ulf Magnusson, Schneider Electric
Brian Meyers, Trane
Jeremy Roberts, LonMark International
Thorsten Roggendorf, Echelon Corporation
Anno Scholten, Individual
John Sublett, Tridium, Inc.
Dave Uden, Trane
Ron Zimmer, Continental Automated Buildings Association (CABA)*
Rob Zivney, Hirsch Electronics Corporation

Appendix B. Revision History

Revision	Date	Editor	Changes Made
wd-0.1	14 Jan 03	Brian Frank	Initial version
wd-0.2	22 Jan 03	Brian Frank	
wd-0.3	30 Aug 04	Brian Frank	Move to Oasis, SysService
wd-0.4	2 Sep 04	Brian Frank	Status
wd-0.5	12 Oct 04	Brian Frank	Namespaces, Writes, Poll
wd-0.6	2 Dec 04	Brian Frank	Incorporate schema comments
wd-0.7	17 Mar 05	Brian Frank	URI, REST, Prototypes, History
wd-0.8	19 Dec 05	Brian Frank	Contracts, Ops
wd-0.9	8 Feb 06	Brian Frank	Watches, Alarming, Bindings
wd-0.10	13 Mar 06	Brian Frank	Overview, XML, clarifications
wd-0.11	20 Apr 06	Brian Frank	10.1 sections, ack, min/max
wd-0.11.1	28 Apr 06	Aaron Hansen	WSDL Corrections
wd-0.12	22 May 06	Brian Frank	Status, feeds, no deltas
wd-0.12.1	29 Jun 06	Brian Frank	Schema, stdlib corrections
obix-1.0-cd-02	30 Jun 06	Aaron Hansen	OASIS document format compliance.
obix-1.0-cs-01	18 Oct 06	Brian Frank	Public review comments
wd-obix.1.1.1	26 Nov 07	Brian Frank	Fixes, date, time, tz
wd-obix.1.1.2	11 Nov 08	Craig Gemmill (from Aaron Hansen)	Add iCalendar scheduling
wd-obix-1.1.3	10 Oct 09	Brian Frank	Remove Scheduling chapter Rev namespace to 1.1 Add Binary Encoding chapter
wd-obix-1.1.4	12 Nov 09	Brian Frank	MUST, SHOULD, MAY History.tz, History.append HTTP Content Negotiation
oBIX-1-1-spec-wd05	01 Jun 10	Toby Considine	Updated to current OASIS Templates, requirements
oBIX-1-1-spec-wd06	08 Jun 10	Brad Benson	Custom facets within binary encoding
oBIX-1-1-spec-wd07	03 Mar 2013	Craig Gemmill	Update to current OASIS templates, fixes
oBIX-1-1-spec-wd08	27 Mar 2013	Craig Gemmill	Changes from feedback

obix-v1.1-wd09	23 Apr 2013	Craig Gemmill	Update to new OASIS template Add of attribute to obix:ref Define additional list semantics Clarify writable w.r.t. add/remove of children Add deletion semantics Add encoding negotiation
obix-v1.1-wd10	08 May 2013	Craig Gemmill	Add CompactHistoryRecord Add preformatted History query Add metadata for alternate hierarchies (tagging)
obix-v1.1-wd11	13 Jun 2013	Craig Gemmill	Modify compact histories per TC feedback
obix-v1.1-wd12	27 Jun 2013	Craig Gemmill	Add delimiter, interval to compact histories
obix-v1.1-wd13	8 July 2013	Toby Considine	Replaced object diagram w/ UML Updated references to other OBIX artifacts
obix-v1.1-CSPRD01	11 July 2013	Paul Knight	Public Review Draft 1
obix-v1.1-wd14	16 Sep 2013	Craig Gemmill	Addressed some comments from PR01; Section 4 rework
obix-v1.1-wd15	30 Sep 2013	Craig Gemmill	Addressed most of PR01 comments
obix-v1.1-wd16	16 Oct 2013	Craig Gemmill	Finished first round of PR01 comments
obix-v1.1-wd17	30 Oct 2013	Craig Gemmill	Reworked Lobby definition, more comments fixed
obix-v1.1-wd18	13 Nov 2013	Craig Gemmill	Added bindings to lobby, oBIX->OBIX
obix-v1.1-wd19	26 Nov 2013	Craig Gemmill	Updated server metadata and Watch sections
obix-v1.1-wd20	4 Dec 2013	Craig Gemmill	WebSockets support for Watches
obix-v1.1-wd21	13 Dec 2013	Craig Gemmill	intermediate revision
obix-v1.1-wd22	17 Dec 2013	Craig Gemmill	More cleanup from JIRA, general Localization added
obix-v1.1-wd23	18 Dec 2013	Craig Gemmill	Replaced UML diagram
obix-v1.1-wd24	19 Dec 2013	Toby Considine	Minor error in Conformance, added bindings to conformance, swapped UML diagram

