



1

2

oBIX Specification

3

Committee Draft 02, June 30, 2006

4

Document identifier:

5

obix-0.12.2

6

Location:

7

<http://www.oasis-open.org/committees/obix>

8

Technical Committee:

9

OASIS Open Building Information Exchange TC

10

Chairs:

11

Toby Considine, University of North Carolina Chapel Hill

12

Paul Ehrlich, Building Intelligence Group

13

Editor:

14

Brian Frank, Tridium

15

Abstract:

16

This document specifies an object model and XML format used for machine-to-machine (M2M) communication.

17

18

Status:

19

This document was last revised or approved by the Open Building Information Exchange TC on the above date. The level of approval is also listed above. Check the current location noted above for possible later revisions of this document. This document is updated periodically on no particular schedule.

20

21

22

23

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 www.oasis-open.org/committees/obix.

24

25

26

27

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 (www.oasis-open.org/committees/obix/ipr.php).

28

29

30

31

The non-normative errata page for this specification is located at www.oasis-open.org/committees/obix.

32

33 Notices

34 OASIS takes no position regarding the validity or scope of any intellectual property or other rights
35 that might be claimed to pertain to the implementation or use of the technology described in this
36 document or the extent to which any license under such rights might or might not be available;
37 neither does it represent that it has made any effort to identify any such rights. Information on
38 OASIS's procedures with respect to rights in OASIS specifications can be found at the OASIS
39 website. Copies of claims of rights made available for publication and any assurances of licenses
40 to be made available, or the result of an attempt made to obtain a general license or permission
41 for the use of such proprietary rights by implementers or users of this specification, can be
42 obtained from the OASIS Executive Director.

43 OASIS invites any interested party to bring to its attention any copyrights, patents or patent
44 applications, or other proprietary rights which may cover technology that may be required to
45 implement this specification. Please address the information to the OASIS Executive Director.

46 Copyright © OASIS Open 2004. All Rights Reserved.

47 This document and translations of it may be copied and furnished to others, and derivative works
48 that comment on or otherwise explain it or assist in its implementation may be prepared, copied,
49 published and distributed, in whole or in part, without restriction of any kind, provided that the
50 above copyright notice and this paragraph are included on all such copies and derivative works.
51 However, this document itself does not be modified in any way, such as by removing the
52 copyright notice or references to OASIS, except as needed for the purpose of developing OASIS
53 specifications, in which case the procedures for copyrights defined in the OASIS Intellectual
54 Property Rights document must be followed, or as required to translate it into languages other
55 than English.

56 The limited permissions granted above are perpetual and will not be revoked by OASIS or its
57 successors or assigns.

58 This document and the information contained herein is provided on an "AS IS" basis and OASIS
59 DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO
60 ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE
61 ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A
62 PARTICULAR PURPOSE.

63 Table of Contents

64	1	Overview	7
65	1.1	XML	7
66	1.2	Networking	7
67	1.3	Normalization	7
68	1.4	Foundation	8
69	2	Quick Start	9
70	3	Architecture	11
71	3.1	Object Model	11
72	3.2	XML	11
73	3.3	URIs	12
74	3.4	REST	12
75	3.5	Contracts	12
76	3.6	Extendibility	13
77	4	Object Model	14
78	4.1	obj	14
79	4.2	bool	15
80	4.3	int	15
81	4.4	real	15
82	4.5	str	15
83	4.6	enum	15
84	4.7	abstime	15
85	4.8	reltime	16
86	4.9	uri	16
87	4.10	list	16
88	4.11	ref	16
89	4.12	err	16
90	4.13	op	16
91	4.14	feed	17
92	4.15	Null	17
93	4.16	Facets	17
94	4.16.1	displayName	17
95	4.16.2	display	17
96	4.16.3	icon	17
97	4.16.4	min	18
98	4.16.5	max	18
99	4.16.6	precision	18
100	4.16.7	range	18
101	4.16.8	status	18
102	4.16.9	unit	19
103	4.16.10	writable	19
104	5	Naming	20
105	5.1	Name	20

106	5.2 Href.....	20
107	5.3 HTTP Relative URIs.....	21
108	5.4 Fragment URIs.....	21
109	6 Contracts.....	22
110	6.1 Contract Terminology.....	22
111	6.2 Contract List.....	22
112	6.3 Is Attribute.....	23
113	6.4 Contract Inheritance.....	23
114	6.5 Override Rules.....	24
115	6.6 Multiple Inheritance.....	25
116	6.6.1 Flattening.....	25
117	6.6.2 Mixins.....	25
118	6.7 Contract Compatibility.....	26
119	6.8 Lists (and Feeds).....	26
120	7 XML.....	28
121	7.1 Design Philosophy.....	28
122	7.2 XML Syntax.....	28
123	7.3 XML Encoding.....	28
124	7.4 XML Decoding.....	29
125	7.5 XML Namespace.....	29
126	7.6 Namespace Prefixes in Contract Lists.....	29
127	8 Operations.....	30
128	9 Object Composition.....	31
129	9.1 Containment.....	31
130	9.2 References.....	31
131	9.3 Extents.....	31
132	9.4 XML.....	32
133	10 Networking.....	33
134	10.1 Request / Response.....	33
135	10.1.1 Read.....	33
136	10.1.2 Write.....	33
137	10.1.3 Invoke.....	34
138	10.2 Errors.....	34
139	10.3 Lobby.....	34
140	10.4 About.....	35
141	10.5 Batch.....	35
142	11 Core Contract Library.....	38
143	11.1 Nil.....	38
144	11.2 Range.....	38
145	11.3 Weekday.....	38
146	11.4 Month.....	38
147	11.5 Units.....	39
148	12 Watches.....	41
149	12.1 WatchService.....	41

150	12.2 Watch	41
151	12.2.1 Watch.add	42
152	12.2.2 Watch.remove	43
153	12.2.3 Watch.pollChanges	43
154	12.2.4 Watch.pollRefresh	43
155	12.2.5 Watch.lease	43
156	12.2.6 Watch.delete	44
157	12.3 Watch Depth	44
158	12.4 Feeds	44
159	13 Points	46
160	13.1 Writable Points	46
161	14 History	47
162	14.1 History Object	47
163	14.2 History Queries	47
164	14.2.1 HistoryFilter	47
165	14.2.2 HistoryQueryOut	48
166	14.2.3 HistoryRecord	48
167	14.2.4 History Query Example	48
168	14.3 History Rollups	49
169	14.3.1 HistoryRollupIn	49
170	14.3.2 HistoryRollupOut	49
171	14.3.3 HistoryRollupRecord	49
172	14.3.4 Rollup Calculation	49
173	14.4 History Feeds	51
174	15 Alarming	52
175	15.1 Alarm States	52
176	15.1.1 Alarm Source	52
177	15.1.2 StatefulAlarm and AckAlarm	52
178	15.2 Alarm Contracts	53
179	15.2.1 Alarm	53
180	15.2.2 StatefulAlarm	53
181	15.2.3 AckAlarm	53
182	15.2.4 PointAlarms	54
183	15.3 AlarmSubject	54
184	15.4 Alarm Feed Example	54
185	16 Security	56
186	16.1 Error Handling	56
187	16.2 Permission based Degradation	56
188	17 HTTP Binding	57
189	17.1 Requests	57
190	17.2 Security	57
191	17.3 Localization	57
192	18 SOAP Binding	59
193	18.1 SOAP Example	59

194	18.2 Error Handling	59
195	18.3 Security	59
196	18.4 Localization	59
197	18.5 WSDL	60
198	Appendix A. Revision History	62
199		

200 1 Overview

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

210

211 The following design points illustrate the problem space oBIX attempts to solve:

- 212 • **XML**: representing M2M information in a standard XML syntax;
- 213 • **Networking**: transferring M2M information in XML over the network;
- 214 • **Normalization**: standard representations for common M2M features: points, histories,
215 and alarms;
- 216 • **Foundation**: providing a common kernel for new standards;

217 1.1 XML

218 The principle requirement of oBIX is to develop a common XML syntax for representing
219 information from diverse M2M systems. The design philosophy of oBIX is based on a small, but
220 extensible data model which maps to a simple fixed XML syntax. This core object model and it's
221 XML syntax is simple enough to capture entirely in one illustration provided in Chapter 4. The
222 object model's extensibility allows for the definition of new abstractions through a concept called
223 *contracts*. The majority of the oBIX specification is actually defined in oBIX itself through
224 contracts.

225 1.2 Networking

226 Once we have a way to represent M2M information in XML, the next step is to provide standard
227 mechanisms to transfer it over networks for publication and consumption. oBIX breaks
228 networking into two pieces: an abstract request/response model and a series of protocol bindings
229 which implement that model. Version 1.0 of oBIX defines two protocol bindings designed to
230 leverage existing web service infrastructure: an HTTP REST binding and a SOAP binding.

231 1.3 Normalization

232 There are a few concepts which have broad applicability in systems which sense and control the
233 physical world. Version 1.0 of oBIX provides a normalized representation for three of these:

- 234 • **Points**: representing a single scalar value and it's status – typically these map to
235 sensors, actuators, or configuration variables like a setpoint;
- 236 • **Histories**: modeling and querying of time sampled point data. Typically edge devices
237 collect a time stamped history of point values which can be fed into higher level
238 applications for analysis;
- 239 • **Alarming**: modeling, routing, and acknowledgment of alarms. Alarms indicate a
240 condition which requires notification of either a user or another application.

241 **1.4 Foundation**

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

248 2 Quick Start

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

256

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

262

263 The first thing to notice is that there are three element types. In oBIX there is a one-to-one
 264 mapping between *objects* and *elements*. Objects are the fundamental abstraction used by the
 265 oBIX data model. Elements are how those objects are expressed in XML syntax. This document
 266 uses the term object and sub-objects, although you can substitute the term element and sub-
 267 element when talking about the XML representation.

268

269 The root `obj` element models the entire thermostat. Its `href` attribute identifies the URI for this
 270 oBIX document. There are three child objects for each of the thermostat’s variables. The `real`
 271 objects store our two floating point values: space temperature and setpoint. The `bool` object
 272 stores a boolean variable for furnace state. Each sub-element contains a `name` attribute which
 273 defines the role within the parent. Each sub-element also contains a `val` attribute for the current
 274 value. Lastly we see that we have annotated the temperatures with an attribute called `units` so
 275 we know they are in Fahrenheit, not Celsius (which would be one hot room). The oBIX
 276 specification defines a bunch of these annotations which are called *facets*.

277

278 In real life, sensor and actuator variables (called *points*) imply more semantics than a simple
 279 scalar value. In other cases such as alarms, it is desirable to standardize a complex data
 280 structure. oBIX captures these concepts into *contracts*. Contracts allow us to tag objects with
 281 normalized semantics and structure.

282

283 Let’s suppose our thermostat’s sensor is reading a value of -412°F? Clearly our thermostat is
 284 busted, so we should report a fault condition. Let’s rewrite the XML to include the status facet
 285 and to provide additional semantics using contracts:

286

```
287 <obj href="http://myhome/thermostat/">
288   <!-- spaceTemp point -->
289   <real name="spaceTemp" is="obix:Point"
290     val="-412.0" status="fault"
291     units="obix:units/fahrenheit"/>
292
293   <!-- setpoint point -->
294   <real name="setpoint" is="obix:Point"
295     val="72.0"
296     unit="obix:units/fahrenheit"/>
297
298 </obj>
```

```
299 <!-- furnaceOn point -->  
300 <bool name="furnaceOn" is="obix:Point" val="true"/>  
301  
302 </obj>
```

303

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

308

309 Contracts play a pivotal role in oBIX for building new abstractions upon the core object model.
310 Contracts are slick because they are just normal objects defined using standard oBIX syntax (see
311 Chapter 13 to take sneak peak the point contracts).

312 3 Architecture

313 The oBIX architecture is based on the following principles:

- 314 • **Object Model:** a concise object model used to define all oBIX information.
- 315 • **XML Syntax:** a simple XML syntax for expressing the object model.
- 316 • **URIs:** URIs are used to identify information within the object model.
- 317 • **REST:** a small set of verbs is used to access objects via their URIs and transfer their
- 318 state via XML.
- 319 • **Contracts:** a template model for expressing new oBIX “types”.
- 320 • **Extendibility:** providing for consistent extendibility using only these concepts.

321 3.1 Object Model

322 All information in oBIX is represented using a small, fixed set of primitives. The base abstraction
323 for these primitives is cleverly called *object*. An object can be assigned a URI and all objects can
324 contain other objects.

325
326 There are eight special kinds of *value objects* used to store a piece of simple information:

- 327 • *bool*: stores a boolean value - true or false;
- 328 • *int*: stores an integer value;
- 329 • *real*: stores a floating point value;
- 330 • *str*: stores a UNICODE string;
- 331 • *enum*: stores an enumerated value within a fixed range;
- 332 • *abstime*: stores an absolute time value (timestamp);
- 333 • *reltime*: stores a relative time value (duration or time span);
- 334 • *uri*: stores a Universal Resource Identifier;

335 Note that any value object can also contain sub-objects. There are also a couple of other special
336 object types: *list*, *op*, *feed*, *ref* and *err*.

337 3.2 XML

338 oBIX is all about a simple XML syntax to represent its underlying object model. Each of the
339 object types map to one type of element. The value objects represent their data value using the
340 *val* attribute. All other aggregation is simply nesting of elements. A simple example to illustrate:

```
341 <obj href="http://bradybunch/people/Mike-Brady/">
342   <obj name="fullName">
343     <str name="first" val="Mike"/>
344     <str name="last" val="Brady"/>
345   </obj>
346   <int name="age" val="45"/>
347   <ref name="spouse" href="/people/Carol-Brady"/>
348   <list name="children">
349     <ref href="/people/Greg-Brady"/>
350     <ref href="/people/Peter-Brady"/>
351     <ref href="/people/Bobby-Brady"/>
352     <ref href="/people/Marsha-Brady"/>
353     <ref href="/people/Jan-Brady"/>
354     <ref href="/people/Cindy-Brady"/>
355   </list>
356 </obj>
```

357 Note in this simple example how the `href` attribute specifies URI references which may be used
 358 to fetch more information about the object. Names and hrefs are discussed in detail in Chapter 5.

359 3.3 URIs

360 No architecture is complete without some sort of naming system. In oBIX everything is an object,
 361 so we need a way to name objects. Since oBIX is really about making information available over
 362 the web using XML, it makes to sense to leverage the venerable URI (Uniform Resource
 363 Identifier). URIs are the standard way to identify “resources” on the web.

364

365 Often URIs also provide information about how to fetch their resource - that’s why they are often
 366 called URLs (Uniform Resource Locator). From a practical perspective if a vendor uses HTTP
 367 URIs to identify their objects, you can most likely just do a simple HTTP GET to fetch the oBIX
 368 document for that object. But technically, fetching the contents of a URI is a protocol binding
 369 issue discussed in later chapters.

370

371 The value of URIs are that they come with all sorts of nifty rules already defined for us (see RFC
 372 3986). For example URIs define which characters are legal and which are illegal. Of great value
 373 to oBIX is *URI references* which define a standard way to express and normalize relative URIs.
 374 Plus most programming environments have libraries to manage URIs so developers don’t have to
 375 worry about nitty gritty normalization details.

376 3.4 REST

377 Many savvy readers may be thinking that objects identified with URIs and passed around as XML
 378 documents is starting to sound a lot like REST – and you would be correct. REST stands for
 379 REpresentational State Transfer and is an architectural style for web services that mimics how
 380 the World Wide Web works. The WWW is basically a big web of HTML documents all
 381 hyperlinked together using URIs. Likewise, oBIX is basically a big web of XML object documents
 382 hyperlinked together using URIs.

383

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

- 388 • **Read:** an object
- 389 • **Write:** an object
- 390 • **Invoke:** an operation

391 3.5 Contracts

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

397

398 In object oriented systems we capture these patterns into classes. In relational databases we
 399 map them into tables with typed columns. In oBIX we capture these patterns using a concept
 400 called *contracts*, which are standard oBIX objects used as a template. Contracts are more nimble
 401 and flexible than strongly typed schema languages, without the overhead of introducing new

402 syntax. A contract document is parsed just like any other oBIX document. In geek speak
403 contracts are a combination of prototype based inheritance and mixins.

404

405 Why do we care about trying to capture these patterns? The most important use of contracts is
406 by the oBIX specification itself to define new standard abstractions. It is just as important for
407 everyone to agree on normalized semantics as it is as on syntax. Contracts also provide the
408 definitions needed to map to the OO guy's classes or the relational database guy's tables.

409 **3.6 Extendibility**

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

415

416 The principle behind oBIX extendibility is that anything new is defined strictly in terms of objects,
417 URIs, and contracts. To put it another way - new abstractions don't introduce any new XML
418 syntax or functionality that client code is forced to care about. New abstractions are always
419 modeled as standard trees of oBIX objects, just with different semantics. That doesn't mean that
420 higher level application code never changes to deal with new abstractions, but the core stack that
421 deals with networking and parsing shouldn't have to change.

422

423 This extendibility model is similar to most mainstream programming languages such as Java or
424 C#. The syntax of the core language is fixed with a built in mechanism to define new
425 abstractions. Extendibility is achieved by defining new class libraries using the language's fixed
426 syntax. This means I don't have to update the compiler every time some one adds a new class.

427

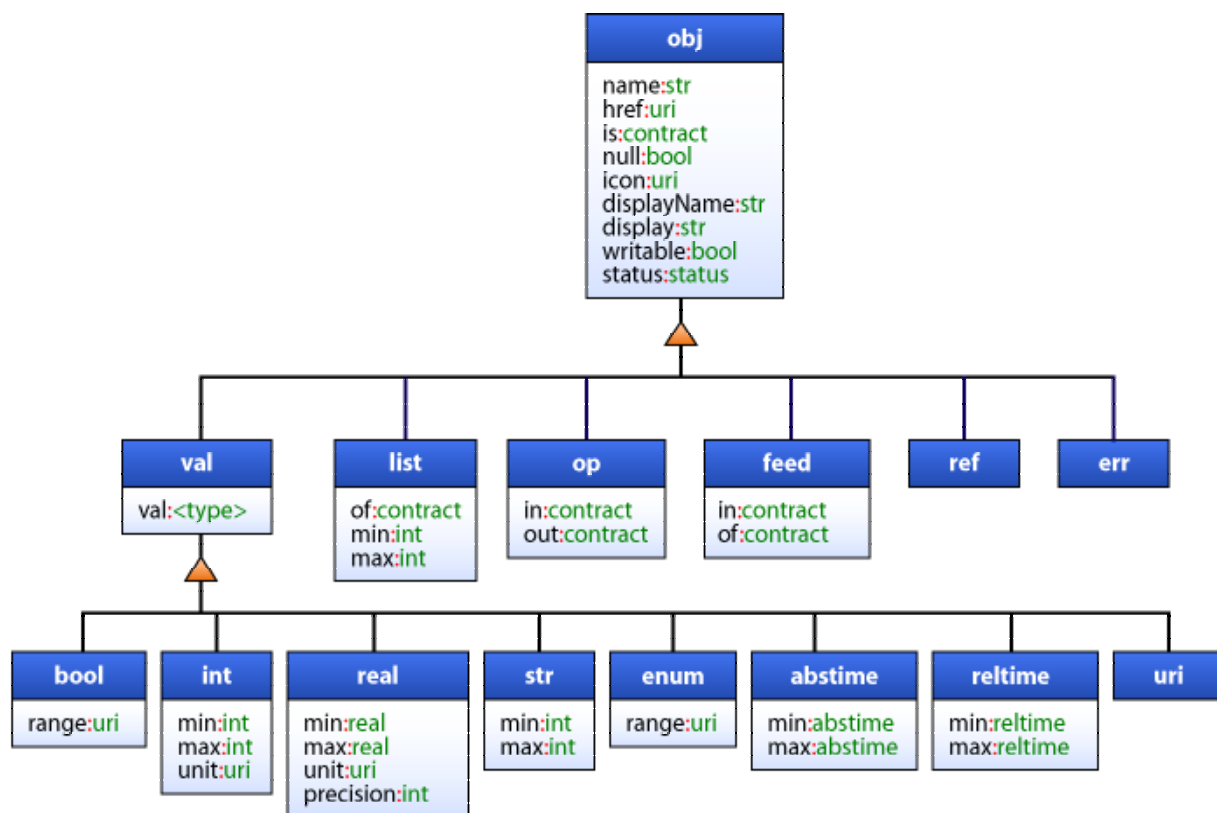
4 Object Model

428

The oBIX specification is based on a small, fixed set of object types. These object types map one to one to an XML element type. The oBIX object model is summarized in the following illustration. Each box represents a specific object type (and XML element name). Each object type also lists its supported attributes.

431

432



433

434

4.1 obj

435

The root abstraction in oBIX is *object*, modeled in XML via the `obj` element. Every XML element in oBIX is a derivative of the `obj` element. Any `obj` element or its derivatives can contain other `obj` elements. The attributes supported on the `obj` element include:

437

438

- **name**: defines the object's purpose in its parent object (discussed in the Chapter 5);

439

- **href**: provides a URI reference for identifying the object (discussed in the Chapter 5);

440

- **is**: defines the contracts the object implements (discussed in Chapter 6);

441

- **null**: support for null objects (discussed in Section 4.15)

442

- **facets**: a set of attributes used to provide meta-data about the object (discussed in Section 4.16);

443

444

- **val**: an attribute used only with value objects (`bool`, `int`, `real`, `str`, `enum`, `abstime`, `reltime`, and `uri`) to store the actual value;

445

The contract definition of `obj`:

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

448 **4.2 bool**

449 The `bool` object represents a boolean condition of either true or false. Its `val` attribute maps to
 450 `xs:boolean` defaulting to false. The literal value of a `bool` must be “true” or “false” (the literals
 451 “1” and “0” are not allowed). The contract definition:

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

453 An example:

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

455 **4.3 int**

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

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

459 An example:

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

461 **4.4 real**

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

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

465 An example:

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

467 **4.5 str**

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

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

471 An example:

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

473 **4.6 enum**

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

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

479 An example:

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

481 **4.7 abstime**

482 The `abstime` type is used to represent an absolute point in time. Its `val` attribute maps to
 483 `xs:dateTime`. Abstimes default to null. The contract definition:

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

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

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

487 **4.8 reltime**

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

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

491 An example of 15 seconds:

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

493 **4.9 uri**

494 The `uri` type is used to store a URI reference. Unlike a plain old `str`, a `uri` has a restricted
495 lexical space as defined by RFC 3986 and XML Schema `anyURI` type. Most URIs will also be a
496 URL, meaning that they identify a resource and how to retrieve it (typically via HTTP). The
497 contract:

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

499 An example for the oBIX home page:

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

501 **4.10 list**

502 The `list` object is a specialized object type for storing a list of other objects. The primary
503 advantage of using a `list` versus a generic `obj` is that lists can specify a common contract for
504 their contents using the `of` attribute. If specified the `of` attribute must be a list of URIs formatted
505 as a contract list. The definition of list is:

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

507 An example list of strings:

```
508 <list of="obix:str">  
509 <str val="one"/>  
510 <str val="two"/>  
511 </list>
```

512 Lists are discussed in greater detail along with contracts in section 6.8.

513 **4.11 ref**

514 The `ref` object is used to create an out of document reference to another oBIX object. It the
515 oBIX equivalent of the HTML anchor tag. The contract definition:

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

517 A `ref` element must always specify a `href` attribute. References are discussed in detail in
518 section 9.2.

519 **4.12 err**

520 The `err` object is a special object used to indicate an error. Its actual semantics are context
521 dependent. Typically `err` objects should include a human readable description of the problem
522 via the `display` attribute. The contract definition:

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

524 **4.13 op**

525 The `op` object is used to define an operation. All operations take one input object as a
526 parameter, and return one object as an output. The input and output contracts are defined via the
527 `in` and `out` attributes. The contract definition:

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


529 Operations are discussed in detail in Chapter 8.

530 4.14 feed

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

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

536 Feeds are subscribed via Watches discussed in Chapter 12.

537 4.15 Null

538 All objects support the concept of *null*. Null is the absence of a value. Null is indicated using the
 539 `null` attribute with a boolean value. All objects default null to false with the exception of `enum`
 540 and `abstime` (since any other default would be confusing).

541

542 Null is inherited from contracts a little differently than other attributes. See Section 6.4 for details.

543 4.16 Facets

544 All objects can be annotated with a predefined set of attributes called *facets*. Facets provide
 545 additional meta-data about the object. The set of available facets is: `displayName`, `display`,
 546 `icon`, `min`, `max`, `precision`, `range`, and `unit`.

547 4.16.1 displayName

548 The `displayName` facet provides a localized human readable name of the object stored as a
 549 `xs:string`:

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

551 Typically the `displayName` facet should be a localized form of the `name` attribute. There are no
 552 restrictions on `displayName` overrides from the contract (although it should be uncommon since
 553 `displayName` is just a human friendly version of `name`).

554 4.16.2 display

555 The `display` facet provides a localized human readable description of the object stored as a
 556 `xs:string`:

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

558 There are no restrictions on `display` overrides from the contract.

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

562 4.16.3 icon

563 The `icon` facet provides a URI reference to a graphical icon which may be used to represent the
 564 object in a user agent:

```
565 <object icon="/icons/equipment.png"/>
```

566 The contents of the `icon` attribute must be a URI to an image file. The image file is preferably a
 567 16x16 PNG file. There are no restrictions on `icon` overrides from the contract.

568 4.16.4 min

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

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

571 The contents of the `min` attribute must match its associated `val` type. The `min` facet is used with
 572 `int`, `real`, `abstime`, and `reltime` to define an inclusive lower limit of the value space. It is
 573 used with `str` to indicate the minimum number of Unicode characters of the string. It is used
 574 with `list` to indicate the minimum number of child objects (named or unnamed). Overrides of
 575 the `min` facet may only narrow the value space using a larger value. The `min` facet must never
 576 be greater than the `max` facet (although they can be equal).

577 4.16.5 max

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

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

580 The contents of the `max` attribute must match its associated `val` type. The `max` facet is used with
 581 `int`, `real`, `abstime`, and `reltime` to define an inclusive upper limit of the value space. It is
 582 used with `str` to indicate the maximum number of Unicode characters of the string. It is used
 583 with `list` to indicate the maximum number of child objects (named or unnamed). Overrides of
 584 the `max` facet may only narrow the value space using a smaller value. The `max` facet must never
 585 be less than the `min` facet (although they can be equal).

586 4.16.6 precision

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

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

589 The contents of the `precision` attribute must be `xs:int`. The value of the `precision`
 590 attribute equates to the number of meaningful decimal places. In the example above, the value of
 591 2 indicates two meaningful decimal places: "75.04". Typically precision is used by client
 592 applications which do their own formatting of `real` values. There are no restrictions on
 593 `precision` overrides.

594 4.16.7 range

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

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

599 The override rule for `range` is that the specified range must inherit from the contract's range.
 600 Enumerations are funny beasts in that specialization of an enum usually involves adding new
 601 items to the range. Technically this is widening the enum's value space, rather than narrowing it.
 602 But in practice, adding items into the range is what we desire.

603 4.16.8 status

604 The `status` facet is used to annotate an object about the quality and state of the information:

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

606 Status is an enumerated string value with one of the following values (ordered by priority):

- 607 • **disabled**: This state indicates that the object has been disabled from normal operation
 608 (out of service). In the case of operations and feeds, this state is used to disable support
 609 for the operation or feed.

- 610 • **fault**: The `fault` state indicates that the data is invalid or unavailable due to a failure
611 condition - data which is out of date, configuration problems, software failures, or
612 hardware failures. Failures involving communications should use the `down` state.
- 613 • **down**: The `down` state indicates a communication failure.
- 614 • **unackedAlarm**: The `unackedAlarm` state indicates there is an existing alarm
615 condition which has not been acknowledged by a user – it is the combination of the
616 alarm and `unacked` states. The difference between `alarm` and `unackedAlarm` is that
617 alarm implies that a user has already acknowledged the alarm or that no human
618 acknowledgement is necessary for the alarm condition. The difference between
619 `unackedAlarm` and `unacked` is that the object has returned to a normal state.
- 620 • **alarm**: This state indicates the object is currently in the alarm state. The alarm state
621 typically means that an object is operating outside of its normal boundaries. In the case
622 of an analog point this might mean that the current value is either above or below its
623 configured limits. Or it might mean that a digital sensor has transitioned to an undesired
624 state. See Alarming (Chapter 15) for additional information.
- 625 • **unacked**: The `unacked` state is used to indicate a past alarm condition which remains
626 unacknowledged.
- 627 • **overridden**: The `overridden` state means the data is ok, but that a local override is
628 currently in effect. An example of an override might be the temporary override of a
629 setpoint from it's normal scheduled setpoint.
- 630 • **ok**: The `ok` state indicates normal status. This is the assumed default state for all
631 objects.

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

635 4.16.9 unit

636 The `unit` facet defines a unit of measurement. A `unit` attribute is a URI reference to a
637 `obix:Unit` object (see section 11.5 for the contract definition). It is used with the `int` and `real`
638 object types:

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

640 It is recommended that the `unit` facet not be overridden if declared in a contract. If it is
641 overridden, then the override should use a `Unit` object with the same dimensions as the contract
642 (it must measure the same physical quantity).

643 4.16.10 writable

644 The `writable` facet specifies if this object can be written by the client. If `false` (the default), then
645 the object is read-only. It is used with all objects except operations and feeds:

```
646       <str name="userName" val="jsmith"       writable="false"/>  

647       <str name="fullName" val="John Smith" writable="true"/>
```

648

649 5 Naming

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

656

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

662

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

669 5.1 Name

670 The name of an object is represented using the `name` attribute. Names are programmatic
671 identifiers with restrictions on their valid character set. A name must contain only ASCII letters,
672 digits, underbar, or dollar signs. A digit may not be used as the first character. Convention is to
673 use camel case with the first character in lower case: "foo", "fooBar", "thisIsOneLongName".
674 Within a given object, all of its direct children must have unique names. Objects which don't
675 have a `name` attribute are called *unnamed objects*. The root object of an oBIX document should
676 not specify a `name` attribute (but almost always has an absolute href URI).

677 5.2 Href

678 The href of an object is represented using the `href` attribute. If specified, the root object must
679 have an absolute URI. All other hrefs within an oBIX document are treated as URI references
680 which may be relative. Because the root href is always an absolute URI, it may be used as the
681 base for normalizing relative URIs within the document. The formal rules for URI syntax and
682 normalization are defined in RFC 3986. We consider a few common cases that serve as design
683 patterns within oBIX in Section 5.3.

684

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

691 5.3 HTTP Relative URIs

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

- 695 • If the URI starts with “*scheme:*” then it is an globally absolute URI
- 696 • If the URI starts with a single slash, then it is server absolute URI
- 697 • If the URI starts with a “#”, then it is a fragment identifier (discussed in next section)
- 698 • If the URI starts with “..”, then the path must backup from the base

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

700 Some examples:

```
701 http://server/a + http://overthere/x → http://overthere/x
702 http://server/a + /x/y/z → http://server/x/y/z
703 http://server/a/b + c → http://server/a/c
704 http://server/a/b/ + c → http://server/a/b/c
705 http://server/a/b + c/d → http://server/a/c/d
706 http://server/a/b/ + c/d → http://server/a/b/c/d
707 http://server/a/b + ../c → http://server/c
708 http://server/a/b/ + ../c → http://server/a/c
```

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

715 5.4 Fragment URIs

716 It is not uncommon to reference an object internal to an oBIX document. This is achieved using
 717 fragment URI references starting with the “#”. Let's consider the example:

```
718 <obj href="http://server/whatever/">
719 <enum name="switch1" range="#onOff" val="on"/>
720 <enum name="switch2" range="#onOff" val="off"/>
721 <list is="obix:Range" href="onOff">
722 <obj name="on"/>
723 <obj name="off"/>
724 </list>
725 </obj>
```

726 In this example there are two objects with a range facet referencing a fragment URI. Any URI
 727 reference starting with “#” must be assumed to reference an object within the same oBIX
 728 document. Clients should not perform another URI retrieval to dereference the object. In this
 729 case the object being referenced is identified via the href attribute.

730

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

```
736 ...
737 <list is="obix:Range" href="#onOff">
738 ...
```

739 6 Contracts

740 Contracts are a mechanism to harness the inherit patterns in modeling oBIX data sources. What
741 is a contract? Well basically it is just a normal oBIX object. What makes a contract object
742 special, is that other objects reference it as a “template object” using the `is` attribute.

743

744 So what does oBIX use contracts for? Contracts solve many problems in oBIX:

- 745 • **Semantics:** contracts are used to define “types” within oBIX. This lets us collectively
746 agree on common object definitions to provide consistent semantics across vendor
747 implementations. For example the `Alarm` contract ensures that client software can
748 extract normalized alarm information from any vendor’s system using the exact same
749 object structure.
- 750 • **Defaults:** contracts also provide a convenient mechanism to specify default values. For
751 example the `Alarm` contract provides a specification for all the default values which don’t
752 need to be passed over the network for every read.
- 753 • **Type Export:** it is likely that many vendors will have a system built using a statically
754 typed language like Java or C#. Contracts provide a standard mechanism to export type
755 information in a format that all oBIX clients can consume.

756

757 Why use contracts versus other approaches? There are certainly lots of ways to solve the above
758 problems. The benefit of the contract design is its flexibility and simplicity. Conceptually
759 contracts provide an elegant model for solving many different problems with one abstraction.
760 From a specification perspective, we can define new abstractions using the oBIX XML syntax
761 itself. And from an implementation perspective, contracts give us a machine readable format that
762 clients already know how to retrieve and parse – to use OO lingo, the exact same syntax is used
763 to represent both a class and an instance.

764 6.1 Contract Terminology

765 In order to discuss contracts, it is useful to define a couple of terms:

- 766 • **Contract:** is a reusable object definition expressed as a standard oBIX XML document.
767 Contracts are the templates or prototypes used as the foundation of the oBIX type
768 system.
- 769 • **Contract List:** is a list of one or more URIs to contract objects. It is used as the value of
770 the `is`, `of`, `in` and `out` attributes. The list of URIs is separated by the space character.
771 You can think of a contract list as a type declaration.
- 772 • **Implements:** when an object specifies a contract in its contract list, the object is said to
773 *implement* the contract. This means that the object is inheriting both the structure and
774 semantics of the specified contract.
- 775 • **Implementation:** an object which implements a contract is said to be an *implementation*
776 of that contract.

777 6.2 Contract List

778 The syntax of a contract list attribute is a list of URI references to other oBIX objects. It is used
779 as the value of the `is`, `of`, `in` and `out` attributes. The URIs within the list are separated by the
780 space character (Unicode 0x20). Just like the `href` attribute, a contract URI can be an absolute

781 URI, server relative, or even a fragment reference. The URIs within a contract list may be scoped
782 with an XML namespace prefix (see Section 7.6).

783 6.3 Is Attribute

784 An object defines the contracts it implements via the `is` attribute. The value of the `is` attribute is
785 a contract list. If the `is` attribute is unspecified, then the following rules are used to determine the
786 implied contract list:

- 787 • If the object is an item inside a `list` or `feed`, then the contract list specified by the `of`
788 attribute is used.
- 789 • If the object overrides (by name) an object specified in one of its contracts, then the
790 contract list of the overridden object is used.
- 791 • If all the above rules fail, then the respective primitive contract is used. For example, an
792 `obj` element has an implied contract of `obix:obj` and `real` an implied contract of
793 `obj:real`.

794 Note that element names such as `bool`, `int`, or `str` are syntactic sugar for an implied contract.
795 However if an object implements one of the primitives, then it must use the correct XML element
796 name. For example if an object implements `obix:int`, then it must be expressed as `<int/>`,
797 rather than `<obj is="obix:int"/>`. Therefore it is invalid to implement multiple value types -
798 such as implementing both `obix:bool` and `obix:int`.

799 6.4 Contract Inheritance

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

- 803 • **Explicit Contract:** defines an object structure which all implementations must conform
804 with.
- 805 • **Implicit Contract:** defines semantics associated with the contract. Usually the implicit
806 contract is documented using natural language prose. It isn’t mathematical, but rather
807 subject to human interpretation.

808 For example when we say an object implements the `Alarm` contract, we immediately know that
809 will have a child called `timestamp`. This structure is in the explicit contract of `Alarm` and is
810 formally defined in XML. But we also attach semantics to what it means to be an `Alarm` object:
811 that the object is providing information about an alarm event. These fuzzy concepts can’t be
812 captured in machine language; rather they can only be captured in prose.

813

814 When an object declares itself to implement a contract it must meet both the explicit contract and
815 the implicit contract. An object shouldn’t put `obix:Alarm` in its contract list unless it really
816 represents an alarm event. There isn’t much more to say about implicit contracts other than it is
817 recommended that a human brain be involved. So now let’s look at the rules governing the
818 explicit contract.

819

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

```
825 <obj href="/def/television">
826   <bool name="power" val="false"/>
827   <int name="channel" val="2" min="2" max="200"/>
```

```

828 </obj>
829
830 <obj href="/livingRoom/tv" is="/def/television">
831   <int name="channel" val="8"/>
832   <int name="volume" val="22"/>
833 </obj>

```

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

838

839 An override is always matched to its contract via the `name` attribute. In the example above we
840 knew we were overriding channel, because we declared an object with a name of `"channel"`. We
841 also declared an object with a name of `"volume"`. Since volume wasn't declared in the contract,
842 we assume it's a new definition specific to this object.

843

844 Also note that the contract's channel object declares a `min` and `max` facet. These two facets are
845 also inherited by the implementation. Almost all attributes are inherited from their contract
846 including facets, `val`, `of`, `in`, and `out`. The `href` attribute are never inherited. The `null`
847 attribute inherits as follows:

- 848 1. If the `null` attribute is specified, then its explicit value is used;
- 849 2. If a `val` attribute is specified and `null` is unspecified, then `null` is implied to be false;
- 850 3. If neither a `val` attribute or a `null` attribute is specified, then the `null` attribute is
851 inherited from the contract;

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

853 6.5 Override Rules

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

858

859 Overriding the explicit contract means to override the value, facets, or contract list. However we
860 can never override the object to be in incompatible value type. For example if the contract
861 specifies a child as `real`, then all implementations must use `real` for that child. As a special
862 case, `obj` may be narrowed to any other element type.

863

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

```

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

```

869 In this example the contract has declared a value space of 2 to 200. Any implementation of this
870 contract must meet this restriction. For example it would an error to override `min` to `-100` since
871 that would widen the value space. However we can narrow the value space by overriding `min` to
872 a number greater than 2 or by overriding `max` to a number less than 200. The specific override
873 rules applicable to each facet are documented in section 4.16.

874 6.6 Multiple Inheritance

875 An object's contract list may specify multiple contract URIs to implement. This is actually quite
876 common - even required in many cases. There are two topics associated with the
877 implementation of multiple contracts:

- 878 • **Flattening:** contract lists must always be *flattened* when specified. This comes into play
879 when a contract has its own contract list (Section 6.6.1).
- 880 • **Mixins:** the mixin design specifies the exact rules for how multiple contracts are merged
881 together. This section also specifies how conflicts are handled when multiple contracts
882 contain children with the same name (Section 6.6.2).

883 6.6.1 Flattening

884 It is common for contract objects themselves to implement contracts, just like it is common in OO
885 languages to chain the inheritance hierarchy. However due to the nature of accessing oBIX
886 documents over a network, we wish to minimize round trip network requests which might be
887 required to "learn" about a complex contract hierarchy. Consider this example:

```
888 <obj href="/A" />
889 <obj href="/B" is="/A" />
890 <obj href="/C" is="/B" />
891 <obj href="/D" is="/C" />
```

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

895

896 Because of these issues, servers are required to flatten their contract inheritance hierarchy into a
897 list when specifying the *is*, *of*, *in*, or *out* attributes. In the example above, the correct
898 representation would be:

```
899 <obj href="/A" />
900 <obj href="/B" is="/A" />
901 <obj href="/C" is="/B /A" />
902 <obj href="/D" is="/C /B /A" />
```

903 This allows clients to quickly scan D's contract list to see that D implements C, B, and A without
904 further requests.

905 6.6.2 Mixins

906 Flattening is not the only reason a contract list might contain multiple contract URIs. oBIX also
907 supports the more traditional notion of multiple inheritance using a mixin metaphor. Consider the
908 following example:

```
909 <obj href="acme:Device">
910 <str name="serialNo"/>
911 </obj>
912
913 <obj href="acme:Clock" is="acme:Device">
914 <op name="snooze">
915 <int name="volume" val="0"/>
916 </obj>
917
918 <obj href="acme:Radio" is="acme:Device">
919 <real name="station" min="87.0" max="107.5">
920 <int name="volume" val="5"/>
921 </obj>
922
923 <obj href="acme:ClockRadio" is="acme:Radio acme:Clock acme:Device"/>
```

924 In this example ClockRadio implements both Clock and Radio. Via flattening of Clock and
925 Radio, ClockRadio also implements Device. In oBIX this is called a *mixin* – Clock, Radio,

926 and `Device` are mixed into (merged into) `ClockRadio`. Therefore `ClockRadio` inherits four
 927 children: `serialNo`, `snooze`, `volume`, and `station`. Mixins are a form of multiple inheritance
 928 akin to Java/C# interfaces (remember oBIX is about the type inheritance, not implementation
 929 inheritance).

930

931 Note that `Clock` and `Radio` both implement `Device` - the classic diamond inheritance pattern.
 932 From `Device`, `ClockRadio` inherits a child named `serialNo`. Furthermore notice that both
 933 `Clock` and `Radio` declare a child named `volume`. This naming collision could potentially create
 934 confusion for what `serialNo` and `volume` mean in `ClockRadio`.

935

936 In oBIX we solve this problem by flattening the contract's children using the following rules:

- 937 1. Process the contract definitions in the order they are listed
- 938 2. If a new child is discovered, it is mixed into the object's definition
- 939 3. If a child is discovered we already processed via a previous contract definition, then the
 940 previous definition takes precedence. However it is an error if the duplicate child is not
 941 *contract compatible* with the previous definition (see Section 6.7).

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

949

950 The first contract in a list is given specific significance since its definition trumps all others. In
 951 oBIX this contract is called the *primary contract*. It is recommended that the primary contract
 952 implement all the other contracts specified in the contract list (this actually happens quite naturally
 953 by itself in many programming languages). This makes it easier for clients to bind the object into
 954 a strongly typed class if desired. Obviously this recommendation doesn't make sense for contract
 955 objects themselves - contracts shouldn't implement themselves.

956 6.7 Contract Compatibility

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

962

963 A contract list X is compatible with contract list Y, if and only if X narrows the value space defined
 964 by Y. This means that X can narrow the set of objects which implement Y, but never expand the
 965 set. Contract compatibility is not commutative (X is compatible with Y does not imply Y is
 966 compatible with X). If that definition sounds too highfaluting, you can boil it down to this practical
 967 rule: X can add new URIs to Y's list, but never any take away.

968 6.8 Lists (and Feeds)

969 Implementations derived from `list` or `feed` contracts inherit the `of` attribute. Like other
 970 attributes we can override the `of` attribute, but only if contract compatible - you must include all of
 971 the URIs in the contract's `of` attribute, but you can add additional ones (see Section 6.7).

972

973 Lists and feeds also have the special ability to implicitly define the contract list of their contents.

974 In the following example it is implied that each child element has a contract list of

975 /def/MissingPerson without actually specifying the `is` attribute in each list item:

976

```
<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>
```

977

978

979

980

981 If an element in the list or feed does specify its own `is` attribute, then it must be contract
982 compatible with the `of` attribute.

983 7 XML

984 Chapter 4 specifies an abstract object model used to standardize how oBIX information is
985 modeled. This chapter specifies how the object model is represented in XML.

986 7.1 Design Philosophy

987 Since there are many different approaches to developing an XML syntax, it is worthwhile to
988 provide a bit of background to how the oBIX XML syntax was designed. Historically in M2M
989 systems, non-standard extensions have been second class citizens at best, but usually opaque.
990 One of the design principles of oBIX is to embrace vertical domain and vendor specific
991 extensions, so that all data and services have a level playing field.

992

993 In order to achieve this goal, the XML syntax is designed to support a small, fixed schema for all
994 oBIX documents. If a client agent understands this very simple syntax, then the client is
995 guaranteed access to the server's object tree regardless of whether those objects implement
996 standard or non-standard contracts.

997

998 Higher level semantics are captured via contracts. Contracts “tag” an object with a type and can
999 be applied dynamically. This is very useful for modeling systems which are dynamically
1000 configured in the field. What is important is that contracts are optionally understood by clients.
1001 Contracts do not effect the XML syntax nor are clients required to use them for basic access to
1002 the object tree. Contracts are merely an abstraction layered cleanly above the object tree and it's
1003 fixed XML syntax.

1004 7.2 XML Syntax

1005 The oBIX XML syntax maps very closely to the abstract object model. The syntax is summarized:

- 1006 • Every oBIX object maps to exactly one XML element;
- 1007 • An object's children are mapped as children XML elements;
- 1008 • The XML element name maps to the built-in object type;
- 1009 • Everything else about an object is represented as XML attributes;

1010 The object model figure in Chapter 4 illustrates the valid XML elements and their respective
1011 attributes.

1012 7.3 XML Encoding

1013 The following rules apply to encoding oBIX documents:

- 1014 • oBIX documents must be well formed XML;
- 1015 • oBIX documents should begin with XML Declaration specifying their encoding ;
- 1016 • It is strongly encouraged to use UTF-8 encoding without a byte order mark;
- 1017 • oBIX documents must not include a Document Type Declaration – oBIX documents
1018 cannot contain an internal or external subset;
- 1019 • oBIX documents should include an XML Namespace definition. Convention is declare
1020 the default namespace of the document to “http://obix.org/ns/schema/1.0”. If oBIX is
1021 embedded inside another type of XML document, then convention is to use “o” as the
1022 namespace prefix. Note that the prefix “obix” should not be used (see Section 7.6).

1023 7.4 XML Decoding

1024 The following rules apply to decoding of oBIX documents:

- 1025 • Must conform to XML processing rules as defined by XML 1.1;
- 1026 • Documents which are not well formed XML must be rejected;
- 1027 • Parsers are not required to understand a Document Type Declaration;
- 1028 • Any unknown element must be ignored regardless of its XML namespace
- 1029 • Any unknown attribute must be ignored regardless of its XML namespace

1030

1031 The basic rule of thumb is: strict in what you generate, and liberal in what you accept. oBIX
 1032 parsers are required to ignore elements and attributes which they do not understand. However
 1033 an oBIX parser should never accept an XML document which isn't well formed (such as
 1034 mismatched tags).

1035 7.5 XML Namespace

1036 XML namespaces for standards within the oBIX umbrella should conform to the following pattern:

1037 `http://obix.org/ns/{spec}/{version}`

1038

1039 The XML namespace for oBIX version 1.0 is:

1040 `http://obix.org/ns/schema/1.0`

1041 All XML in this document is assumed to have this namespace unless otherwise explicitly stated.

1042 7.6 Namespace Prefixes in Contract Lists

1043 XML namespace prefixes defined within an oBIX document may be used to prefix the URIs of a
 1044 contract list. If a URI within a contract list starts with string matching a defined XML prefix
 1045 followed by the ":" colon character, then the URI is normalized by replacing the prefix with it's
 1046 namespace value.

1047

1048 The XML namespace prefix of "obix" is predefined. This prefix is used for all the oBIX defined
 1049 contracts. The "obix" prefix is literally translated into "http://obix.org/def/". For example the URI
 1050 "obix:bool" is translated to "http://obix.org/def/bool". Documents should not define an XML
 1051 namespace using the prefix "obix" which collides with the predefined "obix" prefix – if it is defined,
 1052 then it is superseded by the predefined value of "http://obix.org/def/". All oBIX defined contracts
 1053 are accessible via their HTTP URI using the HTTP binding.

1054

1055 An example oBIX document with XML namespace prefixes normalized:

1056 `<obj xmlns:acme="http://acme.com/def/" is="acme:CustomPoint obix:Point"/>`

1057

1058 `<obj is="http://acme.com/def/CustomPoint http://obix.org/def/Point"/>`

1059 8 Operations

1060 Operations are the things that you can “do” to an oBIX object. They are akin to methods in
1061 traditional OO languages. Typically they map to commands rather than a variable that has
1062 continuous state. Unlike value objects which represent an object and its current state, the `op`
1063 element merely represents the definition of an operation you can invoke.

1064

1065 All operations take exactly one object as a parameter and return exactly one object as a result.
1066 The `in` and `out` attributes define the contract list for the input and output objects. If you need
1067 multiple input or output parameters, then wrap them in a single object using a contract as the
1068 signature. For example:

```
1069 <op href="/addTwoReals" in="/def/AddIn" out="obix:real"/>  
1070  
1071 <obj href="/def/AddIn">  
1072 <real name="a"/>  
1073 <real name="b"/>  
1074 </obj>
```

1075

1076 Objects can override the operation definition from one of their contracts. However the new `in` or
1077 `out` contract list must be contract compatible (see Section 6.7) with the contract’s definition.

1078

1079 If an operation doesn’t require a parameter, then specify `in` as `obix:nil`. If an operation
1080 doesn’t return anything, then specify `out` as `obix:nil`. Occasionally an operation is inherited
1081 from a contract which is unsupported in the implementation. In this case use set the `status`
1082 attribute to `disabled`.

1083

1084 Operations are always invoked via their own `href` attribute (not their parent’s `href`). Therefore
1085 operations should always specify an `href` attribute if you wish clients to invoke them. A common
1086 exception to this rule is contract definitions themselves.

1087 9 Object Composition

1088 A good metaphor for comparison with oBIX is the World Wide Web. If you ignore all the fancy
1089 stuff like JavaScript and Flash, basically the WWW is a web of HTML documents hyperlinked
1090 together with URIs. If you dive down one more level, you could say the WWW is a web of HTML
1091 elements such as `<p>`, `<table>`, and `<div>`.

1092

1093 What the WWW does for HTML documents, oBIX does for objects. The logical model for oBIX is
1094 a global web of oBIX objects linked together via URIs. Some of these oBIX objects are static
1095 documents like contracts or device descriptions. Other oBIX objects expose real-time data or
1096 services. But they all are linked together via URIs to create the *oBIX Web*.

1097

1098 Individual objects are composed together in two ways to define this web. Objects may be
1099 composed together via *containment* or via *reference*.

1100 9.1 Containment

1101 Any oBIX object may contain zero or more children objects. This even includes objects which
1102 might be considered primitives such as `bool` or `int`. All objects are open ended and free to
1103 specify new objects which may not be in the object's contract. Containment is represented in the
1104 XML syntax by nesting the XML elements:

1105

```
1106 <obj href="/a/">
1107   <list name="b" href="b">
1108     <obj href="b/c">
1109   </obj>
1110 </obj>
```

1111

1112 In this example the object identified by `/a` contains `/a/b`, which in turn contains `/a/b/c`. Child
1113 objects may be named or unnamed depending on if the `name` attribute is specified (Section 5.1).
1114 In the example, `/a/b` is named and `/a/b/c` is unnamed. Typically named children are used to
1115 represent fields in a record, structure, or class type. Unnamed children are often used in lists.

1116 9.2 References

1117 Let's go back to our WWW metaphor. Although the WWW is a web of individual HTML elements
1118 like `<p>` and `<div>`, we don't actually pass individual `<p>` elements around over the network.
1119 Rather we "chunk" them into HTML documents and always pass the entire document over the
1120 network. To tie it all together, we create links between documents using the `<a>` anchor
1121 element. These anchors serve as place holders, referencing outside documents via a URI.

1122

1123 A oBIX reference is basically just like an HTML anchor. It serves as placeholder to "link" to
1124 another oBIX object via a URI. While containment is best used to model small trees of data,
1125 references may be used to model very large trees or graphs of objects. As a matter fact, with
1126 references we can link together all oBIX objects on the Internet to create the oBIX Web.

1127 9.3 Extents

1128 When oBIX is applied to a problem domain, we have to decide whether to model relationships
1129 using either containment or references. These decisions have a direct impact on how your model
1130 is represented in XML and accessed over the network. The containment relationship is imbued

1131 with special semantics regarding XML encoding and eventing. In fact, oBIX coins a term for
 1132 containment called an object's *extent*. An object's extent is its tree of children down to
 1133 references. Only objects which have an href have an extent. Objects without an href are always
 1134 included in one or more of their ancestors extents.

1135

```
1136 <obj href="/a/">
1137   <obj name="b" href="b">
1138     <obj name="c"/>
1139     <ref name="d" href="/d"/>
1140   </obj>
1141   <ref name="e" href="/e"/>
1142 </obj>
```

1143

1144 In the example above, we have five objects named 'a' to 'e'. Because 'a' includes an href, it has
 1145 an associated extent, which encompasses 'b' and 'c' by containment and 'd' and 'e' by reference.
 1146 Likewise, 'b' has an href which results in an extent encompassing 'c' by containment and 'd' by
 1147 reference. Object 'c' does not provide a direct href, but exists in both the 'a' and 'b' objects'
 1148 extents. Note an object with an href has exactly one extent, but can be nested inside multiple
 1149 extents.

1150 9.4 XML

1151 When marshaling objects into an XML, it is required that an extent always be fully inlined into the
 1152 XML document. The only valid objects which may be referenced outside the document are `ref`
 1153 element themselves.

1154

1155 If the object implements a contract, then it is required that the extent defined by the contract be
 1156 fully inlined into the document (unless the contract itself defined a child as a `ref` element). An
 1157 example of a contract which specifies a child as a `ref` is `Lobby.about` (10.3).

1158 10 Networking

1159 The heart of oBIX is its object model and associated XML syntax. However, the primary use case
1160 for oBIX is to access information and services over a network. The oBIX architecture is based on
1161 a client/server network model:

- 1162 • **Server:** software containing oBIX enabled data and services. Servers respond to
1163 requests from client over a network.
- 1164 • **Client:** software which makes requests to servers over a network to access oBIX enabled
1165 data and services.

1166 There is nothing to prevent software from being both an oBIX client and server. Although a key
1167 tenant of oBIX is that a client is not required to implement server functionality which might require
1168 a server socket to accept incoming requests.

1169 10.1 Request / Response

1170 All network access is boiled down into three request / response types:

- 1171 • **Read:** return the current state of an object at a given URI as an oBIX XML document.
- 1172 • **Write:** update the state of an existing object at a URI. The state to write is passed over
1173 the network as an oBIX XML document. The new updated state is returned in an oBIX
1174 XML document.
- 1175 • **Invoke:** invoke an operation identified by a given URI. The input parameter and output
1176 result are passed over the network as an oBIX XML document.

1177 Exactly how these three request/responses are implemented between a client and server is
1178 called a *protocol binding*. The oBIX specification defines two standard protocol bindings: HTTP
1179 Binding (see Chapter 17) and SOAP Binding (see Chapter 18). However all protocol bindings
1180 must follow the same read, write, invoke semantics discussed next.

1181 10.1.1 Read

1182 The read request specifies an object's URI and the read response returns the current state of the
1183 object as an oBIX document. The response must include the object's complete extent (see 9.3).
1184 Servers may return an `err` object to indicate the read was unsuccessful – the most common
1185 error is `obix:BadUriErr` (see 10.2 for standard error contracts).

1186 10.1.2 Write

1187 The write request is designed to overwrite the current state of an existing object. The write
1188 request specifies the URI of an existing object and it's new desired state. The response returns
1189 the updated state of the object. The response must include the object's complete extent (see
1190 9.3). The server is free to completely or partially ignore the write, so clients should be prepared
1191 to examine the response to check if the write was successful. Servers may also return an `err`
1192 object to indicate the write was unsuccessful.

1193
1194 Clients are not required to include the object's full extent in the request. Objects explicitly
1195 specified in the request object tree should be overwritten or "overlaid" over the server's actual
1196 object tree. Only the `val` attribute should be specified for a write request (outside of identification
1197 attributes such as `name`). A write operation that provides facets has unspecified behavior. When
1198 writing `int` or `reals` with `units`, the write value must be in the same units as the server

1199 specifies in read requests – clients must not provide a different `unit` facet and expect the server
1200 to auto-convert (in fact the `unit` facet should be not included in the request).

1201 **10.1.3 Invoke**

1202 The invoke request is designed to trigger an operation. The invoke request specified the URI of
1203 an `op` object and the input argument object. The response includes the output object. The
1204 response must include the output object's complete extent (see 9.3). Servers may also return an
1205 `err` object to indicate the invoke was unsuccessful.

1206 **10.2 Errors**

1207 Request errors are conveyed to clients with the `err` element. Any time an oBIX server
1208 successfully receives a request and the request cannot be processed, then the server should
1209 return an `err` object to the client. Returning a valid oBIX document with `err` must be used when
1210 feasible rather than protocol specific error handling (such as an HTTP response code). Such a
1211 design allows for consistency with batch request partial failures and makes protocol binding more
1212 pluggable by separating data transport from application level error handling.

1213

1214 A few contracts are predefined for common errors:

- 1215 • **BadUriErr**: used to indicate either a malformed URI or a unknown URI;
- 1216 • **UnsupportedErr**: used to indicate an a request which isn't supported by the server
1217 implementation (such as an operation defined in a contract, which the server doesn't
1218 support);
- 1219 • **PermissionErr**: used to indicate that the client lacks the necessary security permission
1220 to access the object or operation.

1221 The contracts for these errors are:

```
1222 <err href="obix:BadUriErr"/>
1223 <err href="obix:UnsupportedErr"/>
1224 <err href="obix:PermissionErr"/>
```

1225

1226 If one of the above contracts makes sense for an error, then it should be included in the `err`
1227 element's `is` attribute. It is strongly encouraged to also include a useful description of the
1228 problem in the `display` attribute.

1229 **10.3 Lobby**

1230 All oBIX servers must provide an object which implements `obix:Lobby`. The `Lobby` object
1231 serves as the central entry point into an oBIX server, and lists the URIs for other well-known
1232 objects defined by the oBIX specification. Theoretically all a client needs to know to bootstrap
1233 discovery is one URI for the `Lobby` instance. By convention this URI is "http://server/obix",
1234 although vendors are certainly free to pick another URI. The `Lobby` contract is:

```
1235 <obj href="obix:Lobby">
1236   <ref name="about" is="obix:About"/>
1237   <op name="batch" in="obix:BatchIn" out="obix:BatchOut"/>
1238   <ref name="watchService" is="obix:WatchService"/>
1239 </obj>
```

1240 The `Lobby` instance is where vendors should place vendor specific objects used for data and
1241 service discovery.

1242 10.4 About

1243 The `obix:About` object is a standardized list of summary information about an oBIX server.
 1244 Clients can discover the `About` URI directly from the `Lobby`. The `About` contract is:

```
1245 <obj href="obix:About">
1246
1247   <str name="obixVersion"/>
1248
1249   <str name="serverName"/>
1250   <abstime name="serverTime"/>
1251   <abstime name="serverBootTime"/>
1252
1253   <str name="vendorName"/>
1254   <uri name="vendorUrl"/>
1255
1256   <str name="productName"/>
1257   <str name="productVersion"/>
1258   <uri name="productUrl"/>
1259
1260 </obj>
```

1261
 1262 The following children provide information about the oBIX implementation:

- 1263 • **obixVersion**: specifies which version of the oBIX specification the server implements.
 1264 This string must be a list of decimal numbers separated by the dot character (Unicode
 1265 0x2E).

1266
 1267 The following children provide information about the server itself:

- 1268 • **serverName**: provides a short localized name for the server.
- 1269 • **serverTime**: provides the server's current local time.
- 1270 • **serverBootTime**: provides the server's start time - this should be the start time of the
 1271 oBIX server software, not the machine's boot time.

1272
 1273 The following children provide information about the server's software vendor:

- 1274 • **vendorName**: the company name of the vendor who implemented the oBIX server
 1275 software.
- 1276 • **vendorUrl**: a URI to the vendor's website.

1277
 1278 The following children provide information about the software product running the server:

- 1279 • **productName**: with the product name of oBIX server software.
- 1280 • **productUrl**: a URI to the product's website.
- 1281 • **productVersion**: a string with the product's version number. Convention is to use
 1282 decimal digits separated by dots.

1283 10.5 Batch

1284 The `Lobby` defines a `batch` operation which is used to batch multiple network requests together
 1285 into a single operation. Batching multiple requests together can often provide significant
 1286 performance improvements over individual round-robin network requests. As a general rule, one
 1287 big request will always out-perform many small requests over a network.

1288

1289 A batch request is an aggregation of read, write, and invoke requests implemented as a standard
 1290 oBIX operation. At the protocol binding layer, it is represented as a single invoke request using
 1291 the `Lobby.batch` URI. Batching a set of requests to a server must be processed semantically
 1292 equivalent to invoking each of the requests individually in a linear sequence.

1293

1294 The batch operation inputs a `BatchIn` object and outputs a `BatchOut` object:

```
1295 <list href="obix:BatchIn" of="obix:uri"/>
1296
1297 <list href="obix:BatchOut" of="obix:obj"/>
```

1298

1299 The `BatchIn` contract specifies a list of requests to process identified using the `Read`, `Write`, or
 1300 `Invoke` contract:

```
1301 <uri href="obix:Read"/>
1302
1303 <uri href="obix:Write">
1304   <obj name="in"/>
1305 </uri>
1306
1307 <uri href="obix:Invoke">
1308   <obj name="in"/>
1309 </uri>
```

1310

1311 The `BatchOut` contract specifies an ordered list of the response objects to each respective
 1312 request. For example the first object in `BatchOut` must be the result of the first request in
 1313 `BatchIn`. Failures are represented using the `err` object. Every `uri` passed via `BatchIn` for a
 1314 read or write request must have a corresponding result `obj` in `BatchOut` with an `href` attribute
 1315 using an identical string representation from `BatchIn` (no normalization or case conversion is
 1316 allowed).

1317

1318 It is up to vendors to decide how to deal with partial failures. In general idempotent requests
 1319 should indicate a partial failure using `err`, and continue processing additional requests in the
 1320 batch. If a server decides not to process additional requests when an error is encountered, then
 1321 it is still required to return an `err` for each respective request not processed.

1322

1323 Let's look at a simple example:

1324

```
1325 <list is="obix:BatchIn">
1326   <uri is="obix:Read" val="/someStr"/>
1327   <uri is="obix:Read" val="/invalidUri"/>
1328   <uri is="obix:Write" val="/someStr">
1329     <str name="in" val="new string value"/>
1330   </uri>
1331 </list>
1332
1333 <list is="obix:BatchOut">
1334   <str href="/someStr" val="old string value"/>
1335   <err href="/invalidUri" is="obix:BadUriErr" display="href not found"/>
1336   <str href="/someStr" val="new string value"/>
1337 </list>
```

1338

1339 In this example, the batch request is specifying a read request for `"/someStr"` and `"/invalidUri"`,
 1340 followed by a write request to `"/someStr"`. Note that the write request includes the value to write
 1341 as a child named `"in"`.

1342

1343 The server responds to the batch request by specifying exactly one object for each request URI.
1344 The first read request returns a `str` object indicating the current value identified by `"/someStr"`.
1345 The second read request contains an invalid URI, so the server returns an `err` object indicating a
1346 partial failure and continues to process subsequent requests. The third request is a write to
1347 `"someStr"`. The server updates the value at `"someStr"`, and returns the new value. Note that
1348 because the requests are processed in order, the first request provides the original value of
1349 `"someStr"` and the third request contains the new value. This is exactly what we would expect
1350 had we processed each of these requests individually.

1351 11 Core Contract Library

1352 This chapter defines some fundamental object contracts that serve as building blocks for the oBIX
1353 specification.

1354 11.1 Nil

1355 The `obix:nil` contract defines a standardized null object. Nil is commonly used for an
1356 operation's `in` or `out` attribute to denote the absence of an input or output. The definition:

```
1357 <obj href="obix:nil" null="true"/>
```

1358 11.2 Range

1359 The `obix:Range` contract is used to define an `bool` or `enum`'s range. Range is a list object that
1360 contains zero or more objects called the range items. Each item's `name` attribute specifies the
1361 identifier used as the literal value of an `enum`. Item ids are never localized, and must be used
1362 only once in a given range. You may use the optional `displayName` attribute to specify a
1363 localized string to use in a user interface. The definition of Range:

```
1364 <list href="obix:Range" of="obix:obj"/>
```

1365 An example:

```
1366 <list href="/enums/OffSlowFast" is="obix:Range">  
1367   <obj name="off" displayName="Off"/>  
1368   <obj name="slow" displayName="Slow Speed"/>  
1369   <obj name="fast" displayName="Fast Speed"/>  
1370 </list>
```

1371 The range facet may be used to define the localized text of a `bool` value using the ids of "true"
1372 and "false":

```
1373 <list href="/enums/OnOff" is="obix:Range">  
1374   <obj name="true" displayName="On"/>  
1375   <obj name="false" displayName="Off"/>  
1376 </list >
```

1377 11.3 Weekday

1378 The `obix:Weekday` contract is a standardized enum for the days of the week:

```
1379 <enum href="obix:Weekday" range="#Range">  
1380   <list href="#Range" is="obix:Range">  
1381     <obj name="sunday" />  
1382     <obj name="monday" />  
1383     <obj name="tuesday" />  
1384     <obj name="wednesday" />  
1385     <obj name="thursday" />  
1386     <obj name="friday" />  
1387     <obj name="saturday" />  
1388   </list>  
1389 </enum>
```

1390 11.4 Month

1391 The `obix:Month` contract is a standardized enum for the months of the year:

```
1392 <enum href="obix:Month" range="#Range">  
1393   <list href="#Range" is="obix:Range">  
1394     <obj name="january" />  
1395     <obj name="february" />  
1396     <obj name="march" />  
1397     <obj name="april" />
```

```

1398     <obj name="may" />
1399     <obj name="june" />
1400     <obj name="july" />
1401     <obj name="august" />
1402     <obj name="september" />
1403     <obj name="october" />
1404     <obj name="november" />
1405     <obj name="december" />
1406 </list>
1407 </enum>

```

1408 11.5 Units

1409 Representing units of measurement in software is a thorny issue. oBIX provides a unit framework
 1410 for mathematically defining units within the object model. An extensive database of predefined
 1411 units is also provided.

1412

1413 All units measure a specific quantity or dimension in the physical world. Most known dimensions
 1414 can be expressed as a ratio of the seven fundamental dimensions: length, mass, time,
 1415 temperature, electrical current, amount of substance, and luminous intensity. These seven
 1416 dimensions are represented in SI respectively as kilogram (kg), meter (m), second (sec), Kelvin
 1417 (K), ampere (A), mole (mol), and candela (cd).

1418

1419 The `obix:Dimension` contract defines the ratio of the seven SI units using a positive or
 1420 negative exponent:

```

1421 <obj href="obix:Dimension">
1422   <int name="kg" val="0"/>
1423   <int name="m" val="0"/>
1424   <int name="sec" val="0"/>
1425   <int name="K" val="0"/>
1426   <int name="A" val="0"/>
1427   <int name="mol" val="0"/>
1428   <int name="cd" val="0"/>
1429 </obj>

```

1430 A `Dimension` object contains zero or more ratios of kg, m, sec, K, A, mol, or cd. Each of these
 1431 ratio maps to the exponent of that base SI unit. If a ratio is missing then the default value of zero
 1432 is implied. For example acceleration is m/s^2 , which would be encoded in oBIX as:

```

1433 <obj is="obix:Dimension">
1434   <int name="m" val="1"/>
1435   <int name="sec" val="-2"/>
1436 </obj>

```

1437

1438 Units with equal dimensions are considered to measure the same physical quantity. This is not
 1439 always precisely true, but is good enough for practice. This means that units with the same
 1440 dimension are convertible. Conversion can be expressed by specifying the formula required to
 1441 convert the unit to the dimension's normalized unit. The normalized unit for every dimension is
 1442 the ratio of SI units itself. For example the normalized unit of energy is the joule $m^2 \cdot kg \cdot s^{-2}$. The
 1443 kilojoule is 1000 joules and the watt-hour is 3600 joules. Most units can be mathematically
 1444 converted to their normalized unit and to other units using the linear equations:

```

1445 unit = dimension • scale + offset
1446 toNormal = scalar • scale + offset
1447 fromNormal = (scalar - offset) / scale
1448 toUnit = fromUnit.fromNormal( toUnit.toNormal(scalar) )

```

1449 There are some units which don't fit this model including logarithm units and units dealing with
 1450 angles. But this model provides a practical solution for most problem spaces. Units which don't
 1451 fit this model should use a dimension where every exponent is set to zero. Applications should
 1452 not attempt conversions on these types of units.

1453

1454 The `obix:Unit` contract defines a unit including its dimension and its `toNormal` equation:

```
1455 <obj href="obix:Unit">
1456   <str name="symbol"/>
1457   <obj name="dimension" is="obix:Dimension"/>
1458   <real name="scale" val="1"/>
1459   <real name="offset" val="0"/>
1460 </obj>
```

1461 The unit element contains a `symbol`, `dimension`, `scale`, and `offset` sub-object:

- 1462 • **symbol**: The `symbol` element defines a short abbreviation to use for the unit. For
1463 example “°F” would be the symbol for degrees Fahrenheit. The `symbol` element should
1464 always be specified.
- 1465 • **dimension**: The `dimension` object defines the dimension of measurement as a ratio of
1466 the seven base SI units. If omitted, the `dimension` object defaults to the
1467 `obix:Dimension` contract, in which case the ratio is the zero exponent for all seven
1468 base units.
- 1469 • **scale**: The `scale` element defines the scale variable of the `toNormal` equation. The
1470 `scale` object defaults to 1.
- 1471 • **offset**: The `offset` element defines the offset variable of the `toNormal` equation. If
1472 omitted then `offset` defaults to 0.

1473 The `display` attribute should be used to provide a localized full name for the unit based on the
1474 client's locale. If the `display` attribute is omitted, clients should use `symbol` for display
1475 purposes.

1476

1477 An example for the predefined unit for kilowatt:

```
1478 <obj href="obix:units/kilowatt" display="kilowatt">
1479   <str name="symbol" val="kW"/>
1480   <obj name="dimension">
1481     <int name="m" val="2"/>
1482     <int name="kg" val="1"/>
1483     <int name="sec" val="-3"/>
1484   </obj>
1485   <real name="scale" val="1000"/>
1486 </obj>
```

1487

1488 Automatic conversion of units is considered a localization issue – see Section 17.3 for more
1489 details.

1490 12 Watches

1491 A key requirement of oBIX is access to real-time information. We wish to enable clients to
 1492 efficiently receive access to rapidly changing data. However, we don't want to require clients to
 1493 implement web servers or expose a well-known IP address. In order to address this problem,
 1494 oBIX provides a model for client polled eventing called *watches*. The watch lifecycle is as follows:

- 1495 • The client creates a new watch object with the `make` operation on the server's
 1496 `WatchService` URI. The server defines a new `Watch` object and provides a URI to
 1497 access the new watch.
- 1498 • The client registers (and unregisters) objects to watch using operations on the `Watch`
 1499 object.
- 1500 • The client periodically polls the `Watch` URI using the `pollChanges` operation to obtain
 1501 the events which have occurred since the last poll.
- 1502 • The server frees the `Watch` under two conditions. The client may explicitly free the
 1503 `Watch` using the `delete` operation. Or the server may automatically free the `Watch`
 1504 because the client fails to poll after a predetermined amount of time (called the lease
 1505 time).

1506

1507 Watches allow a client to maintain a real-time cache for the current state of one or more objects.
 1508 They are also used to access an event stream from a `feed` object. Plus, watches serve as the
 1509 standardized mechanism for managing per-client state on the server via leases.

1510 12.1 WatchService

1511 The `WatchService` object provides a well-known URI as the factory for creating new watches.
 1512 The `Watch` URI is available directly from the `Lobby` object. The contract for `WatchService`:

```
1513 <obj href="obix:WatchService">
1514   <op name="make" in="obix:Nil" out="obix:Watch"/>
1515 </obj>
```

1516 The `make` operation returns a new empty `Watch` object as an output. The href of the newly
 1517 created `Watch` object can then be used for invoking operations to populate and poll the data set.

1518 12.2 Watch

1519 `Watch` object is used to manage a set of objects which are subscribed and periodically polled by
 1520 clients to receive the latest events. The contract is:

```
1521 <obj href="obix:Watch">
1522   <retime name="lease" min="PT0S" writable="true"/>
1523   <op name="add" in="obix:WatchIn" out="obix:WatchOut"/>
1524   <op name="remove" in="obix:WatchIn"/>
1525   <op name="pollChanges" out="obix:WatchOut"/>
1526   <op name="pollRefresh" out="obix:WatchOut"/>
1527   <op name="delete"/>
1528 </obj>
1529
1530 <obj href="obix:WatchIn">
1531   <list name="hrefs" of="obix:WatchInItem"/>
1532 </obj>
1533
1534 <uri href="obix:WatchInItem">
1535   <obj name="in"/>
1536 </uri>
1537
```

```

1538 <obj href="obix:WatchOut">
1539   <list name="values" of="obix:obj"/>
1540 </obj>

```

1541

1542 Many of the Watch operations use two contracts: `obix:WatchIn` and `obix:WatchOut`. The
 1543 client identifies objects to `add` and `remove` from the poll list via `WatchIn`. This object contains a
 1544 list of URIs. Typically these URIs should be server relative.

1545

1546 The server responds to `add`, `pollChanges`, and `pollRefresh` operations via the `WatchOut`
 1547 contract. This object contains the list of subscribed objects - each object must specify an `href`
 1548 URI using the exact same string as the URI identified by the client in the corresponding `WatchIn`.
 1549 Servers are not allowed to perform any case conversions or normalization on the URI passed by
 1550 the client. This allows client software to use the URI string as a hash key to match up server
 1551 responses.

1552 **12.2.1 Watch.add**

1553 Once a Watch has been created, the client can add new objects to watch using the `add`
 1554 operation. This operation inputs a list of URIs and outputs the current value of the objects
 1555 referenced. The objects returned are required to specify an `href` using the exact string
 1556 representation input by the client. If any object cannot be processed, then a partial failure should
 1557 be expressed by returning an `err` object with the respective `href`. Subsequent URIs must not be
 1558 effected by the failure of one invalid URI. The `add` operation should never return objects not
 1559 explicitly included in the input URIs (even if there are already existing objects in the watch list).
 1560 No guarantee is made that the order of objects in `WatchOut` match the order in of URIs in
 1561 `WatchIn` - clients must use the URI as a key for matching.

1562

1563 Note that the URIs supplied via `WatchIn` may include an optional `in` parameter. This parameter
 1564 is only used when subscribing a watch to a `feed` object. Feeds also differ from other objects in
 1565 that they return a list of historic events in `WatchOut`. Feeds are discussed in detail in Section
 1566 12.4.

1567

1568 It is invalid to add an `op`'s `href` to a watch, the server must report an `err`.

1569

1570 If an attempt is made to add a URI to a watch which was previously already added, then the
 1571 server should return the current object's value in the `WatchOut` result, but treat poll operations as
 1572 if the URI was only added once - polls should only return the object once. If an attempt is made
 1573 to add the same URI multiple times in the same `WatchIn` request, then the server should only
 1574 return the object once.

1575

1576 Note: the lack of a trailing slash can cause problems with watches. Consider a client which adds
 1577 a URI to a watch without a trailing slash. The client will use this URI as a key in its local
 1578 hashtable for the watch. Therefore the server must use the URI exactly as the client specified.
 1579 However, if the object's extent includes children objects they will not be able to use relative URIs. It
 1580 is recommended that servers fail-fast in these cases and return a `BadUriErr` when clients attempt
 1581 to add a URI without a trailing slash to a watch (even though they may allow it for a normal read
 1582 request).

1583 **12.2.2 Watch.remove**

1584 The client can remove objects from the watch list using the `remove` operation. A list of URIs is
1585 input to `remove`, and the `Nil` object is returned. Subsequent `pollChanges` and `pollRefresh`
1586 operations must cease to include the specified URIs. It is possible to remove every URI in the
1587 watch list; but this scenario must not automatically free the `Watch`, rather normal poll and lease
1588 rules still apply. It is invalid to use the `WatchInItem.in` parameter for a `remove` operation.

1589 **12.2.3 Watch.pollChanges**

1590 Clients should periodically poll the server using the `pollChanges` operation. This operation
1591 returns a list of the subscribed objects which have changed. Servers should only return the
1592 objects which have been modified since the last poll request for the specific `Watch`. As with `add`,
1593 every object must specify an `href` using the exact same string representation the client passed in
1594 the original `add` operation. The entire extent of the object should be returned to the client if any
1595 one thing inside the extent has changed on the server side.

1596

1597 Invalid URIs must never be included in the response (only in `add` and `pollRefresh`). An
1598 exception to this rule is when an object which is valid is removed from the URI space. Servers
1599 should indicate an object has been removed via an `err` with the `BadUriErr` contract.

1600 **12.2.4 Watch.pollRefresh**

1601 The `pollRefresh` operation forces an update of every object in the watch list. The server must
1602 return every object and its full extent in the response using the `href` with the exact same string
1603 representation passed by the client in the original `add`. Invalid URIs in the poll list should be
1604 included in the response as an `err` element. A `pollRefresh` resets the poll state of every
1605 object, so that the next `pollChanges` only returns objects which have changed state since the
1606 `pollRefresh` invocation.

1607 **12.2.5 Watch.lease**

1608 All `Watches` have a *lease time*, specified by the `lease` child. If the lease time elapses without
1609 the client initiating a request on the `Watch`, then the server is free to *expire* the watch. Every new
1610 poll request resets the lease timer. So as long as the client polls at least as often as the lease
1611 time, the server should maintain the `Watch`. The following requests should reset the lease timer:
1612 read of the `Watch` URI itself or invocation of the `add`, `remove`, `pollChanges`, or `pollRefresh`
1613 operations.

1614

1615 Clients may request a difference lease time by writing to the `lease` object (requires servers to
1616 assign an `href` to the `lease` child). The server is free to honor the request, cap the lease within a
1617 specific range, or ignore the request. In all cases the write request will return a response
1618 containing the new lease time in effect.

1619

1620 Servers should report expired watches by returning an `err` object with the `BadUriErr` contract.
1621 As a general principle servers should honor watches until the lease runs out or the client explicitly
1622 invokes `delete`. However, servers are free to cancel watches as needed (such as power failure)
1623 and the burden is on clients to re-establish a new watch.

1624 12.2.6 Watch.delete

1625 The `delete` operation can be used to cancel an existing watch. Clients should always delete
 1626 their watch when possible to be good oBIX citizens. However servers should always cleanup
 1627 correctly without an explicit delete when the lease expires.

1628 12.3 Watch Depth

1629 When a watch is put on an object which itself has children objects, how does a client know how
 1630 "deep" the subscription goes? oBIX requires watch depth to match an object's extent (see
 1631 Section 9.3). When a watch is put on a target object, a server must notify the client of any
 1632 changes to any of the objects within that target object's extent. If the extent includes `feed`
 1633 objects they are not included in the watch – feeds have special watch semantics discussed in
 1634 Section 12.4. This means a watch is inclusive of all descendents within the extent except `refs`
 1635 and `feeds`.

1636 12.4 Feeds

1637 Servers may expose event streams using the `feed` object. The event instances are typed via the
 1638 feed's `of` attribute. Clients subscribe to events by adding the feed's href to a watch, optionally
 1639 passing an input parameter which is typed via the feed's `in` attribute. The object returned from
 1640 `Watch.add` is a list of historic events (or the empty list if no event history is available).
 1641 Subsequent calls to `pollChanges` returns the list of events which have occurred since the last
 1642 poll.

1643

1644 Let's consider a simple example for an object which fires an event when its geographic location
 1645 changes:

```
1646 <obj href="/car/">
1647   <feed href="moved" of="/def/Coordinate"/>
1648 </obj>
1649
1650 <obj href="/def/Coordinate">
1651   <real name="lat"/>
1652   <real name="long"/>
1653 </obj>
```

1654

1655 We subscribe to the moved event feed by adding `/car/moved` to a watch. The `WatchOut` will
 1656 include the list of any historic events which have occurred up to this point in time. If the server
 1657 does not maintain an event history this list will be empty:

```
1658 <obj is="obix:WatchIn">
1659   <list names="hrefs"/>
1660     <uri val="/car/moved" />
1661   </list>
1662 </obj>
1663
1664 <obj is="obix:WatchOut">
1665   <list names="values">
1666     <feed href="/car/moved" of="/def/Coordinate/" /> <!-- empty history -->
1667   </list>
1668 </obj>
```

1669

1670 Now every time we call `pollChanges` for the watch, the server will send us the list of event
 1671 instances which have accumulated since our last poll:

```
1672 <obj is="obix:WatchOut">
1673   <list names="values">
1674     <feed href="/car/moved" of="/def/Coordinate">
1675     </obj>
```

```
1676     <real name="lat" val="37.645022"/>
1677     <real name="long" val="-77.575851"/>
1678   </obj>
1679   <obj>
1680     <real name="lat" val="37.639046"/>
1681     <real name="long" val="-77.61872"/>
1682   </obj>
1683 </feed>
1684 </list>
1685 </obj>
```

1686

1687 Note the feed's `of` attribute works just like the `list`'s `of` attribute. The children event instances
1688 are assumed to inherit the contract defined by `of` unless explicitly overridden. If an event
1689 instance does override the `of` contract, then it must be contract compatible. Refer to the rules
1690 defined in Section 6.8.

1691

1692 Invoking a `pollRefresh` operation on a watch with a feed that has an event history, should
1693 return all the historical events as if the `pollRefresh` was an `add` operation. If an event history
1694 is not available, then `pollRefresh` should act like a normal `pollChanges` and just return the
1695 events which have occurred since the last poll.

1696 13Points

1697 Anyone familiar with automation systems immediately identifies with the term *point* (sometimes
 1698 called *tags* in the industrial space). Although there are many different definitions, generally points
 1699 map directly to a sensor or actuator (called *hard points*). Sometimes the concept of a point is
 1700 mapped to a configuration variable such as a software setpoint (called *soft points*). In some
 1701 systems point is an atomic value, and in others it encapsulates a whole truckload of status and
 1702 configuration information.

1703

1704 The goal of oBIX is to capture a normalization representation of points without forcing an
 1705 impedance mismatch on vendors trying to make their native system oBIX accessible. To meet
 1706 this requirement, oBIX defines a low level abstraction for point - simply one of the primitive value
 1707 types with associated status information. Point is basically just a marker contract used to tag an
 1708 object as exhibiting "point" semantics:

```
1709 <obj href="obix:Point"/>
```

1710

1711 This contract must only be used with the value primitive types: `bool`, `real`, `enum`, `str`,
 1712 `abstime`, and `reltime`. Points should use the `status` attribute to convey quality information.
 1713 The following table specifies how to map common control system semantics to a value type:

bool	digital point	<code><bool is="obix:Point" val="true"/></code>
real	analog point	<code><real is="obix:Point" val="22" units="obix:units/celsius"/></code>
enum	multi-state point	<code><enum is="obix:Point" val="slow"/></code>

1714 13.1 Writable Points

1715 Different control systems handle point writes using a wide variety of semantics. Sometimes we
 1716 write a point at a specific priority level. Sometimes we override a point for a limited period of time,
 1717 after which the point falls back to a default value. The oBIX specification doesn't attempt to
 1718 impose a specific model on vendors. Rather oBIX provides a standard `WritablePoint` contract
 1719 which may be extended with additional mixins to handle special cases. `WritablePoint` defines
 1720 write as an operation which takes a `WritePointIn` structure containing the value to write. The
 1721 contracts are:

```
1722 <obj href="obix:WritablePoint" is="obix:Point">
1723   <op name="writePoint" in="obix:WritePointIn" out="obix:Point"/>
1724 </obj>
1725
1726 <obj href="obix:WritePointIn">
1727   <obj name="value"/>
1728 </obj>
```

1729

1730 It is implied that the value passed to `writePoint` match the type of the point. For example if
 1731 `WritablePoint` is used with an `enum`, then `writePoint` must pass an `enum` for the value.

1732 14 History

1733 Most automation systems have the ability to persist periodic samples of point data to create a
 1734 historical archive of a point's value over time. This feature goes by many names including logs,
 1735 trends, or histories. In oBIX, a *history* is defined as a list of time stamped point values. The
 1736 following features are provided by oBIX histories:

- 1737 • **History Object**: a normalized representation for a history itself;
- 1738 • **History Record**: a record of a point sampling at a specific timestamp
- 1739 • **History Query**: a standard way to query history data as Points;
- 1740 • **History Rollup**: a standard mechanism to do basic rollups of history data;

1741 14.1 History Object

1742 Any object which wishes to expose itself as a standard oBIX history implements the
 1743 obix:History contract:

```
1744 <obj href="obix:History">
1745   <int name="count" min="0" val="0"/>
1746   <abstime name="start" null="true"/>
1747   <abstime name="end" null="true"/>
1748   <op name="query" in="obix:HistoryFilter" out="obix:HistoryQueryOut"/>
1749   <feed name="feed" in="obix:HistoryFilter" of="obix:HistoryRecord"/>
1750   <op name="rollup" in="obix:HistoryRollupIn" out="obix:HistoryRollupOut"/>
1751 </obj>
```

1752 Let's look at each of History's sub-objects:

- 1753 • **count**: this field stores the number of history records contained by the history;
- 1754 • **start**: this field provides the timestamp of the oldest record;
- 1755 • **end**: this field provides the timestamp of the newest record;
- 1756 • **query**: the query object is used to query the history to read history records;
- 1757 • **feed**: used to subscribe to a real-time feed of history records;
- 1758 • **rollup**: this object is used to perform history rollups (it is only supported for numeric
 1759 history data);

1760

1761 An example of a history which contains an hour of 15 minute temperature data:

```
1762 <obj href="http://x/outsideAirTemp/history/" is="obix:History">
1763   <int name="count" val="5"/>
1764   <abstime name="start" val="2005-03-16T14:00"/>
1765   <abstime name="end" val="2005-03-16T15:00"/>
1766   <op name="query" href="query"/>
1767   <op name="rollup" href="rollup"/>
1768 </obj>
```

1769 14.2 History Queries

1770 Every History object contains a query operation to query the historical data.

1771 14.2.1 HistoryFilter

1772 The History.query input contract:

```
1773 <obj href="obix:HistoryFilter">
1774   <int name="limit" null="true"/>
```

```

1775 <abstime name="start" null="true"/>
1776 <abstime name="end" null="true"/>
1777 </obj>

```

1778 These fields are described in detail:

- 1779 • **limit**: an integer indicating the maximum number of records to return. Clients can use
- 1780 this field to throttle the amount of data returned by making it non-null. Servers must
- 1781 never return more records than the specified limit. However servers are free to return
- 1782 fewer records than the limit.
- 1783 • **start**: if non-null this field indicates an inclusive lower bound for the query's time range.
- 1784 • **end**: if non-null this field indicates an inclusive upper bound for the query's time range.

1785 14.2.2 HistoryQueryOut

1786 The `History.query` output contract:

```

1787 <obj href="obix:HistoryQueryOut">
1788 <int name="count" min="0" val="0"/>
1789 <abstime name="start" null="true"/>
1790 <abstime name="end" null="true"/>
1791 <list name="data" of="obix:HistoryRecord"/>
1792 </obj>

```

1793 Just like `History`, every `HistoryQueryOut` returns `count`, `start`, and `end`. But unlike

1794 `History`, these values are for the query result, not the entire history. The actual history data is

1795 stored as a list of `HistoryRecords` in the `data` field. Remember that child order is not

1796 guaranteed in oBIX, therefore it might be common to have `count` after `data`.

1797 14.2.3 HistoryRecord

1798 The `HistoryRecord` contract specifies a record in a history query result:

```

1799 <obj href="obix:HistoryRecord">
1800 <abstime name="timestamp" null="true"/>
1801 <obj name="value" null="true"/>
1802 </obj>

```

1803 Typically the value should be on the value types used with `obix:Point`.

1804 14.2.4 History Query Example

1805 An example query from the `/outsideAirTemp/history` example above:

```

1806 <obj href="http://x/outsideAirTemp/history/query" is="obix:HistoryQueryOut">
1807 <int name="count" val="5">
1808 <abstime name="start" val="2005-03-16T14:00"/>
1809 <abstime name="end" val="2005-03-16T15:00"/>
1810 <list name="data" of="#RecordDef obix:HistoryRecord">
1811 <obj> <abstime name="timestamp" val="2005-03-16T14:00"/>
1812 <real name="value" val="40"/> </obj>
1813 <obj> <abstime name="timestamp" val="2005-03-16T14:15"/>
1814 <real name="value" val="42"/> </obj>
1815 <obj> <abstime name="timestamp" val="2005-03-16T14:30"/>
1816 <real name="value" val="43"/> </obj>
1817 <obj> <abstime name="timestamp" val="2005-03-16T14:45"/>
1818 <real name="value" val="47"/> </obj>
1819 <obj> <abstime name="timestamp" val="2005-03-16T15:00"/>
1820 <real name="value" val="44"/> </obj>
1821 </list>
1822 <obj href="#RecordDef" is="obix:HistoryRecord">
1823 <real name="value" units="obix:units/fahrenheit"/>
1824 </obj>
1825 </obj>

```

1826 Note in the example above how the `data` list uses a document local contract to define facets

1827 common to all the records (although we still have to flatten the contract list).

1828 14.3 History Rollups

1829 Control systems collect historical data as raw time sampled values. However, most applications
 1830 wish to consume historical data in a summarized form which we call *rollups*. The rollup operation
 1831 is used summarize an interval of time. History rollups only apply to histories which store numeric
 1832 information as a list of *RealPoints*. Attempting to query a rollup on a non-numeric history such
 1833 as a history of *BoolPoints* should result in an error.

1834 14.3.1 HistoryRollupIn

1835 The `History.rollup` input contract extends `HistoryFilter` to add an interval parameter:

```
1836 <obj href="obix:HistoryRollupIn" is="obix:HistoryFilter">
1837   <reltime name="interval"/>
1838 </obj>
```

1839 14.3.2 HistoryRollupOut

1840 The `History.rollup` output contract:

```
1841 <obj href="obix:HistoryRollupOut">
1842   <int name="count" min="0" val="0"/>
1843   <abstime name="start" null="true"/>
1844   <abstime name="end" null="true"/>
1845   <list name="data" of="obix:HistoryRollupRecord"/>
1846 </obj>
```

1847 The `HistoryRollupOut` object looks very much like `HistoryQueryOut` except it returns a list
 1848 of `HistoryRollupRecords`, rather than `HistoryRecords`. Note: unlike `HistoryQueryOut`,
 1849 the `start` for `HistoryRollupOut` is exclusive, not inclusive. This issue is discussed in greater
 1850 detail next.

1851 14.3.3 HistoryRollupRecord

1852 A history rollup returns a list of `HistoryRollupRecords`:

```
1853 <obj href="obix:HistoryRollupRecord">
1854   <abstime name="start"/>
1855   <abstime name="end" />
1856   <int name="count"/>
1857   <real name="min" />
1858   <real name="max" />
1859   <real name="avg" />
1860   <real name="sum" />
1861 </obj>
```

1862 The children are defined as:

- 1863 • **start**: the exclusive start time of the record's rollup interval;
- 1864 • **end**: the inclusive end time of the record's rollup interval;
- 1865 • **count**: the number of records used to compute this rollup interval;
- 1866 • **min**: specifies the minimum value of all the records within the interval;
- 1867 • **max**: specifies the maximum value of all the records within the interval;
- 1868 • **avg**: specifies the mathematical average of all the values within the interval;
- 1869 • **sum**: specifies the summation of all the values within the interval;

1870 14.3.4 Rollup Calculation

1871 The best way to understand how rollup calculations work is through an example. Let's consider a
 1872 history of meter data where we collected two hours of 15 minute readings of kilowatt values:

```

1873 <obj is="obix:HistoryQueryOut">
1874   <int name="count" val="9">
1875   <abstime name="start" val="2005-03-17T12:00"/>
1876   <abstime name="end" val="2005-03-17T14:00"/>
1877   <list name="data" of="#HistoryDef obix:HistoryRecord">
1878     <obj> <abstime name="timestamp" val="2005-03-17T12:00"/>
1879       <real name="value" val="80"> </obj>
1880     <obj> <abstime name="timestamp" val="2005-03-17T12:15"/>
1881       <real name="value" val="82"></obj>
1882     <obj> <abstime name="timestamp" val="2005-03-17T12:30"/>
1883       <real name="value" val="90"> </obj>
1884     <obj> <abstime name="timestamp" val="2005-03-17T12:45"/>
1885       <real name="value" val="85"> </obj>
1886     <obj> <abstime name="timestamp" val="2005-03-17T13:00"/>
1887       <real name="value" val="81"> </obj>
1888     <obj> <abstime name="timestamp" val="2005-03-17T13:15"/>
1889       <real name="value" val="84"> </obj>
1890     <obj> <abstime name="timestamp" val="2005-03-17T13:30"/>
1891       <real name="value" val="91"> </obj>
1892     <obj> <abstime name="timestamp" val="2005-03-17T13:45"/>
1893       <real name="value" val="83"> </obj>
1894     <obj> <abstime name="timestamp" val="2005-03-17T14:00"/>
1895       <real name="value" val="78"> </obj>
1896   </list>
1897   <obj href="#HistoryRecord" is="obix:HistoryRecord">
1898     <real name="value" units="obix:units/kilowatt"/>
1899   </obj>
1900 </obj>

```

1901

1902 If we were to query the rollup using an interval of 1 hour with a start time of 12:00 and end time of
 1903 14:00, the result should be:

```

1904 <obj is="obix:HistoryRollupOut obix:HistoryQueryOut">
1905   <int name="count" val="2">
1906   <abstime name="start" val="2005-03-17T12:00"/>
1907   <abstime name="end" val="2005-03-17T14:00"/>
1908   <list name="data" of="obix:HistoryRollupRecord">
1909     <obj>
1910       <abstime name="start" val="2005-03-16T12:00"/>
1911       <abstime name="end" val="2005-03-16T13:00"/>
1912       <int name="count" val="4" />
1913       <real name="min" val="81" />
1914       <real name="max" val="90" />
1915       <real name="avg" val="84.5" />
1916       <real name="sum" val="338" />
1917     </obj>
1918     <obj>
1919       <abstime name="start" val="2005-03-16T13:00"/>
1920       <abstime name="end" val="2005-03-16T14:00"/>
1921       <int name="count" val="4" />
1922       <real name="min" val="78" />
1923       <real name="max" val="91" />
1924       <real name="avg" val="84" />
1925       <real name="sum" val="336" />
1926     </obj>
1927   </list>
1928 </obj>

```

1929 If you whip out your calculator, the first thing you will note is that the first raw record of 80kW was
 1930 never used in the rollup. This is because start time is always exclusive. The reason start time
 1931 has to be exclusive is because we are summarizing discrete samples into a contiguous time
 1932 range. It would be incorrect to include a record in two different rollup intervals! To avoid this
 1933 problem we always make start time exclusive and end time inclusive. The following table
 1934 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
------------------	------------------	-------------------------

1935 **14.4 History Feeds**

1936 The `History` contract specifies a feed for subscribing to a real-time feed of the history records.
1937 `History.feed` reuses the same `HistoryFilter` input contract used by `History.query` –
1938 the same semantics apply. When adding a History feed to a watch, the initial result should
1939 contain the list of `HistoryRecords` filtered by the input parameter (the initial result should
1940 match what `History.query` would return). Subsequent calls to `Watch.pollChanges` should
1941 return any new `HistoryRecords` which have been collected since the last poll that also satisfy
1942 the `HistoryFilter`.

1943 15 Alarming

1944 The oBIX alarming feature specifies a normalized model to query, watch, and acknowledge
 1945 alarms. In oBIX, an alarm indicates a condition which requires notification of either a user or
 1946 another application. In many cases an alarm requires acknowledgement, indicating that
 1947 someone (or something) has taken action to resolve the alarm condition. The typical lifecycle of
 1948 an alarm is:

- 1949 1. **Source Monitoring:** algorithms in a server monitor an *alarm source*. An alarm source is
 1950 an object with an href which has the potential to generate an alarm. Example of alarm
 1951 sources might include sensor points (this room is too hot), hardware problems (disk is
 1952 full), or applications (building is consuming too much energy at current energy rates)
- 1953 2. **Alarm Generation:** if the algorithms in the server detect that an alarm source has
 1954 entered an alarm condition, then an *alarm* record is generated. Every alarm is uniquely
 1955 identified using an href and represented using the `obix:Alarm` contract. Sometimes we
 1956 refer to the alarm transition as *off-normal*.
- 1957 3. **To Normal:** many alarm sources are said to be *stateful* - eventually the alarm source
 1958 exits the alarm state, and is said to return *to-normal*. Stateful alarms implement the
 1959 `obix:StatefulAlarm` contract. When the source transitions to normal, we update
 1960 `normalTimestamp` of the alarm.
- 1961 4. **Acknowledgement:** often we require that a user or application acknowledges that they
 1962 have processed an alarm. These alarms implement the `obix:AckAlarm` contract.
 1963 When the alarm is acknowledged, we update `ackTimestamp` and `ackUser`.

1964 15.1 Alarm States

1965 Alarm state is summarized with two variables:

- 1966 • **In Alarm:** is the alarm source currently in the alarm condition or in the normal condition.
 1967 This variable maps to the `alarm` status state.
- 1968 • **Acknowledged:** is the alarm acknowledged or unacknowledged. This variable maps to
 1969 the `unacked` status state.

1970
 1971 Either of these states may transition independent of the other. For example an alarm source can
 1972 return to normal before or after an alarm has been acknowledged. Furthermore it is not
 1973 uncommon to transition between normal and off-normal multiple times generating several alarm
 1974 records before any acknowledgements occur.

1975
 1976 Note not all alarms have state. An alarm which implements neither `StatefulAlarm` nor the
 1977 `AckAlarm` contracts is completely stateless – these alarms merely represent event. An alarm
 1978 which implements `StatefulAlarm` but not `AckAlarm` will have an in-alarm state, but not
 1979 acknowledgement state. Conversely an alarm which implements `AckAlarm` but not
 1980 `StatefulAlarm` will have an acknowledgement state, but not in-alarm state.

1981 15.1.1 Alarm Source

1982 The current alarm state of an alarm source is represented using the `status` attribute. This
 1983 attribute is discussed in Section 4.16.8. It is recommended that alarm sources always report their
 1984 status via the `status` attribute.

1985 15.1.2 StatefulAlarm and AckAlarm

1986 An `Alarm` record is used to summarize the entire lifecycle of an alarm event. If the alarm
 1987 implements `StatefulAlarm` it tracks transition from off-normal back to normal. If the alarm

1988 implements `AckAlarm`, then it also summarizes the acknowledgement. This allows for four
 1989 discrete alarm states:

alarm	acked	normalTimestamp	ackTimestamp
true	false	null	null
true	true	null	non-null
false	false	non-null	null
false	true	non-null	non-null

1990 15.2 Alarm Contracts

1991 15.2.1 Alarm

1992 The core Alarm contract is:

```
1993 <obj href="obix:Alarm">
1994 <ref name="source"/>
1995 <abstime name="timestamp"/>
1996 </obj>
```

1997
 1998 The child objects are:

- 1999 • **source**: the URI which identifies the alarm source. The source should reference an oBIX
 2000 object which models the entity that generated the alarm.
- 2001 • **timestamp**: this is the time at which the alarm source transitioned from normal to off-
 2002 normal and the Alarm record was created.

2003 15.2.2 StatefulAlarm

2004 Alarms which represent an alarm state which may transition back to normal must implement the
 2005 StatefulAlarm contract:

```
2006 <obj href="obix:StatefulAlarm" is="obix:Alarm">
2007 <abstime name="normalTimestamp" null="true"/>
2008 </obj>
```

2009
 2010 The child object is:

- 2011 • **normalTimestamp**: if the alarm source is still in the alarm condition, then this field is null.
 2012 Otherwise this indicates the time of the transition back to the normal condition.

2013 15.2.3 AckAlarm

2014 Alarms which support acknowledgement must implement the `AckAlarm` contract:

```
2015 <obj href="obix:AckAlarm" is="obix:Alarm">
2016 <abstime name="ackTimestamp" null="true"/>
2017 <str name="ackUser" null="true"/>
2018 <op name="ack" in="obix:AlarmAckIn" out="obix:AlarmAckOut"/>
2019 </obj>
2020
2021 <obj href="obix:AckAlarmIn">
2022 <str name="ackUser" null="true"/>
2023 </obj>
2024
2025 <obj href="obix:AckAlarmOut">
2026 <obj name="alarm" is="obix:AckAlarm obix:Alarm"/>
2027 </obj>
```

2028
 2029 The child objects are:

- 2030 • **ackTimestamp**: if the alarm is unacknowledged, then this field is null. Otherwise this
2031 indicates the time of the acknowledgement.
- 2032 • **ackUser**: if the alarm is unacknowledged, then this field is null. Otherwise this field
2033 should provide a string indicating who was responsible for the acknowledgement.
2034

2035 The `ack` operation is used to programmatically acknowledge the alarm. The client may optionally
2036 specify an `ackUser` string via `AlarmAckIn`. However, the server is free to ignore this field
2037 depending on security conditions. For example a highly trusted client may be allowed to specify
2038 its own `ackUser`, but a less trustworthy client may have its `ackUser` predefined based on the
2039 authentication credentials of the protocol binding. The `ack` operation returns an `AckAlarmOut`
2040 which contains the updated alarm record. Use the `Lobby.batch` operation to efficiently
2041 acknowledge a set of alarms.

2042 15.2.4 PointAlarms

2043 It is very common for an alarm source to be an `obix:Point`. A respective `PointAlarm`
2044 contract is provided as a normalized way to report the value which caused the alarm condition:

```
2045 <obj href="obix:PointAlarm" is="obix:Alarm">
2046   <obj name="alarmValue"/>
2047 </obj>
```

2048 The `alarmValue` object should be one of the value types defined for `obix:Point` in Section
2049 13.

2050 15.3 AlarmSubject

2051 Servers which implement oBIX alarming must provide one or more objects which implement the
2052 `AlarmSubject` contract. The `AlarmSubject` contract provides the ability to categorize and
2053 group the sets of alarms a client may discover, query, and watch. For instance a server could
2054 provide one `AlarmSubject` for all alarms and other `AlarmSubjects` based on priority or time
2055 of day. The contract for `AlarmSubject` is:

```
2056 <obj href="obix:AlarmSubject">
2057   <int name="count" min="0" val="0"/>
2058   <op name="query" in="obix:AlarmFilter" out="obix:AlarmQueryOut"/>
2059   <feed name="feed" in="obix:AlarmFilter" of="obix:Alarm"/>
2060 </obj>
2061
2062 <obj href="obix:AlarmFilter">
2063   <int name="limit" null="true"/>
2064   <abstime name="start" null="true"/>
2065   <abstime name="end" null="true"/>
2066 </obj>
2067
2068 <obj href="obix:AlarmQueryOut">
2069   <int name="count" min="0" val="0"/>
2070   <abstime name="start" null="true"/>
2071   <abstime name="end" null="true"/>
2072   <list name="data" of="obix:Alarm"/>
2073 </obj>
```

2074
2075 The `AlarmSubject` follows the same design pattern as `History`. The `AlarmSubject`
2076 specifies the active count of alarms; however, unlike `History` it does not provide the `start`
2077 and `end` bounding timestamps. It contains a `query` operation to read the current list of alarms
2078 with an `AlarmFilter` to filter by time bounds. `AlarmSubject` also contains a `feed` object
2079 which may be used to subscribe to the alarm events.

2080 15.4 Alarm Feed Example

2081 The following example illustrates how a feed works with this `AlarmSubject`:

```

2082 <obj is="obix:AlarmSubject" href="/alarms/">
2083   <int name="count" val="2"/>
2084   <op name="query" href="query"/>
2085   <feed name="feed" href="feed" />
2086 </obj>

```

2087

2088 The server indicates it has two open alarms under the specified AlarmSubject. If a client were
 2089 to add the AlarmSubject's feed to a watch:

```

2090 <obj is="obix:WatchIn">
2091   <list names="hrefs"/>
2092     <uri val="/alarms/feed" />
2093   </list>
2094 </obj>
2095
2096 <obj is="obix:WatchOut">
2097   <list names="values">
2098     <feed href="/alarms/feed" of="obix:Alarm">
2099       <obj href="/alarmdb/528" is="obix:StatefulAlarm obix:PointAlarm obix:Alarm">
2100         <ref name="source" href="/airHandlers/2/returnTemp"/>
2101         <abstime name="timestamp" val="2006-05-18T14:20"/>
2102         <abstime name="normalTimestamp" null="null"/>
2103         <real name="alarmValue" val="80.2"/>
2104       </obj>
2105       <obj href="/alarmdb/527" is="obix:StatefulAlarm obix:PointAlarm obix:Alarm">
2106         <ref name="source" href="/doors/frontDoor"/>
2107         <abstime name="timestamp" val="2006-05-18T14:18"/>
2108         <abstime name="normalTimestamp" null="null"/>
2109         <real name="alarmValue" val="true"/>
2110       </obj>
2111     </feed>
2112   </list>
2113 </obj>

```

2114

2115 The watch returns the historic list of alarm events which is two open alarms. The first alarm
 2116 indicates an out of bounds condition in AirHandler-2's return temperature. The second alarm
 2117 indicates that the system has detected that the front door has been propped open.

2118

2119 Now let's fictionalize that the system detects the front door is closed, and alarm point transitions
 2120 to the normal state. The next time the client polls the watch the alarm would show up in the feed
 2121 list (along with any additional changes or new alarms not shown here):

```

2122 <obj is="obix:WatchOut">
2123   <list names="values">
2124     <obj href="/alarmdb/527" is="obix:StatefulAlarm obix:PointAlarm obix:Alarm">
2125       <ref name="source" href="/doors/frontDoor"/>
2126       <abstime name="timestamp" val="2006-05-18T14:18"/>
2127       <abstime name="normalTimestamp" null="2006-05-18T14:45"/>
2128       <real name="alarmValue" val="true"/>
2129     </obj>
2130   </feed>
2131 </list>
2132 </obj>

```

2133 16Security

2134 Security is a broad topic, that covers many issues:

- 2135 • **Authentication:** verifying a user (client) is who he says he is;
- 2136 • **Encryption:** protecting oBIX documents from prying eyes;
- 2137 • **Permissions:** checking a user's permissions before granting access to read/write objects
2138 or invoke operations;
- 2139 • **User Management:** managing user accounts and permissions levels;

2140

2141 The basic philosophy of oBIX is to leave these issues outside of the specification. Authentication
2142 and encryption is left as a protocol binding issue. Privileges and user management is left as a
2143 vendor implementation issue. Although it is entirely possible to define a publicly exposed user
2144 management model through oBIX, this specification does not define any standard contracts for
2145 user management.

2146 16.1 Error Handling

2147 It is expected that an oBIX server will perform authentication and utilize those user credentials for
2148 checking permissions before processing read, write, and invoke requests. As a general rule,
2149 servers should return `err` with the `obix:PermissionErr` contract to indicate a client lacks the
2150 permission to perform a request. In particularly sensitive applications, a server may instead
2151 choose to return `BadUriErr` so that an untrustworthy client is unaware that a specific object
2152 even exists.

2153 16.2 Permission based Degradation

2154 Servers should strive to present their object model to a client based on the privileges available to
2155 the client. This behavior is called *permission based degradation*. The following rules summarize
2156 effective permission based degradation:

- 2157 1. If an object cannot be read, then it should not be discoverable through objects which are
2158 available.
- 2159 2. Servers should attempt to group standard contracts within the same privilege level – for
2160 example don't split `obix:History's start` and `end` into two different security levels
2161 such that a client might be able to read `start`, and not `end`.
- 2162 3. Don't include a contract in an object's `is` attribute if the contract's children are not
2163 readable to the client.
- 2164 4. If an object isn't writable, then make sure the `writable` attribute is set to `false` (either
2165 explicitly or through a contract default).
- 2166 5. If an `op` inherited from a visible contract cannot be invoked, then set the `null` attribute to
2167 `true` to disable it.

2168 17 HTTP Binding

2169 The HTTP binding specifies a simple REST mapping of oBIX requests to HTTP. A read request
2170 is a simple HTTP GET, which means that you can simply read an object by typing its URI into
2171 your browser. Refer to “RFC 2616 Hypertext Transfer Protocol” for the full specification of HTTP
2172 1.1.

2173 17.1 Requests

2174 The following table summarizes how oBIX requests map to HTTP methods:

oBIX Request	HTTP Method	Target
Read	GET	Any object with an href
Write	PUT	Any object with an href and writable=true
Invoke	POST	Any op object

2175 The URI used for an HTTP request must map to the URI of the object being read, written, or
2176 invoked. Read requests use a simple HTTP GET and return the resulting oBIX document. Write
2177 and invoke are implemented with the PUT and POST methods respectively. The input is passed
2178 to the server as an oBIX document and the result is returned as an oBIX document.

2179

2180 If the oBIX server processes a request, then it must return the resulting oBIX document with an
2181 HTTP status code of 200 OK. The 200 status code must be used even if the request failed and
2182 the server is returning an `err` object as the result.

2183

2184 The oBIX documents passed between client and servers should specify a MIME type of “text/xml”
2185 for the Content-Type HTTP header.

2186

2187 Clients and servers must encode the oBIX document passed over the network using standard
2188 XML encoding rules. It is strongly recommended using UTF8 without a byte-order mark. If
2189 specified, the Content-Encoding HTTP header must match the XML encoding.

2190 17.2 Security

2191 Numerous standards are designed to provide authentication and encryption services for HTTP.
2192 Existing standards should be used when applicable for oBIX HTTP implementations including:

- 2193 • RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication
- 2194 • RFC 2818 - HTTP Over TLS (HTTPS)
- 2195 • RFC 4346/2246 – The TLS Protocol (Transport Layer Security)

2196

2197 17.3 Localization

2198 Servers should localize appropriate data based on the desired locale of the client agent.
2199 Localization should include the `display` and `displayName` attributes. The desired locale of the
2200 client should be determined through authentication or via the Accept-Language HTTP header. A
2201 suggested algorithm is to check if the authenticated user has a preferred locale configured in the

2202 server's user database, and if not then fallback to the locale derived from the Accept-Language
2203 header.
2204
2205 Localization may include auto-conversion of units. For example if the authenticated user has a
2206 configured a preferred unit system such as English versus Metric, then the server might attempt
2207 to convert values with an associated `unit` facet to the desired unit system.

2208 18SOAP Binding

2209 The SOAP binding maps a SOAP operation to each of the three oBIX request types: read, write
 2210 and invoke. Like the HTTP binding, read is supported by every object, write is supported by
 2211 objects whose `writable` attribute is `true`, and invoke is only supported by operations. Inputs
 2212 and outputs of each request are specific to the target object.

2213

2214 Unlike the HTTP binding, requests are not accessed via the URI of the target object, but instead
 2215 via the URI of the SOAP server with the object's URI encoded into the body of the SOAP
 2216 envelope.

2217 18.1 SOAP Example

2218 The following is a SOAP request to an oBIX server's `About` object:

```
2219 <env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
2220 <env:Body>
2221 <read xmlns="http://obix.org/ns/wsd/1.0"
2222 href="http://localhost/obix/about" />
2223 </env:Body>
2224 </env:Envelope>
```

2225

2226 An example response to the above request:

```
2227 <env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
2228 <env:Body>
2229 <obj name="about"
2230 href="http://localhost/obix/about/"
2231 xmlns="http://obix.org/ns/schema/1.0">
2232 <str name="obixVersion" val="1.0"/>
2233 <str name="serverName" val="obix"/>
2234 <abstime name="serverTime" val="2006-02-08T09:40:55.000+05:00"/>
2235 <abstime name="serverBootTime" val="2006-02-08T09:33:31.980+05:00"/>
2236 <str name="vendorName" val="Acme, Inc."/>
2237 <uri name="vendorUrl" val="http://www.acme.com"/>
2238 <str name="productName" val="Acme oBIX Server"/>
2239 <str name="productVersion" val="1.0.3"/>
2240 <uri name="productUrl" val="http://www.acme.com/obix"/>
2241 </obj>
2242 </env:Body>
2243 </env:Envelope>
```

2244 18.2 Error Handling

2245 The oBIX specification defines no SOAP faults. If a request is processed by an oBIX server, then
 2246 a valid oBIX document should be returned with a failure indicated via the `err` object.

2247 18.3 Security

2248 Refer to the recommendations in WS-I Basic Profile 1.0 for security:

2249 <http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html#security>

2250 18.4 Localization

2251 SOAP bindings should follow localization patterns defined for the HTTP binding when applicable
 2252 (see Section 17.3).

2253 18.5 WSDL

2254 In the types section of the WSDL document, the oBIX schema is imported. Server
2255 implementations might consider providing the `schemaLocation` attribute which is absent in the
2256 standard document.

2257

2258 Missing from the standard oBIX WSDL is the service element. This element binds a SOAP server
2259 instance with a network address. Each instance will have to provide its own services section of
2260 the WSDL document. The following is an example of the WSDL service element:

2261

2262

2263

2264

2265

2266

2267

```
<wsdl:service name="obix">
  <wsdl:port name="obixPort" binding="tns:obixSoapBinding">
    <soap:address location="http://localhost/obix/soap"/>
  </wsdl:port>
</wsdl:service>
```

2268

Standard oBIX WSDL is:

2269

2270

2271

2272

2273

2274

2275

2276

2277

2278

2279

2280

2281

2282

2283

2284

2285

2286

2287

2288

2289

2290

2291

2292

2293

2294

2295

2296

2297

2298

2299

2300

2301

2302

2303

2304

2305

2306

2307

2308

2309

2310

2311

2312

2313

2314

2315

2316

2317

```
<wsdl:definitions targetNamespace="http://obix.org/ns/wsd/1.0"
  xmlns="http://obix.org/ns/wsd/1.0"
  xmlns:wsd="http://schemas.xmlsoap.org/wsd/"
  xmlns:soap="http://schemas.xmlsoap.org/wsd/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:obix="http://obix.org/ns/schema/1.0">
  <wsdl:types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified"
      targetNamespace="http://obix.org/ns/wsd/1.0">
      <xsd:import namespace="http://obix.org/ns/schema/1.0"/>
      <xsd:complexType name="ReadReq">
        <xsd:attribute name="href" type="xsd:anyURI"/>
      </xsd:complexType>
      <xsd:complexType name="WriteReq">
        <xsd:complexContent>
          <xsd:extension base="ReadReq">
            <xsd:sequence>
              <xsd:element ref="obix:obj" maxOccurs="1" minOccurs="1"/>
            </xsd:sequence>
          </xsd:extension>
        </xsd:complexContent>
      </xsd:complexType>
      <xsd:complexType name="InvokeReq">
        <xsd:complexContent>
          <xsd:extension base="ReadReq">
            <xsd:sequence>
              <xsd:element ref="obix:obj" maxOccurs="1" minOccurs="1"/>
            </xsd:sequence>
          </xsd:extension>
        </xsd:complexContent>
      </xsd:complexType>
      <xsd:element name="read" type="ReadReq"/>
      <xsd:element name="write" type="WriteReq"/>
      <xsd:element name="invoke" type="InvokeReq"/>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="readSoapReq">
    <wsdl:part name="body" element="read"/>
  </wsdl:message>
  <wsdl:message name="readSoapRes">
    <wsdl:part name="body" element="obix:obj"/>
  </wsdl:message>
  <wsdl:message name="writeSoapReq">
    <wsdl:part name="body" element="write"/>
  </wsdl:message>
  <wsdl:message name="writeSoapRes">
    <wsdl:part name="body" element="obix:obj"/>
  </wsdl:message>
```

```

2318 <wsdl:message name="invokeSoapReq">
2319   <wsdl:part name="body" element="invoke"/>
2320 </wsdl:message>
2321 <wsdl:message name="invokeSoapRes">
2322   <wsdl:part name="body" element="obix:obj"/>
2323 </wsdl:message>
2324 <wsdl:portType name="oBIXSoapPort">
2325   <wsdl:operation name="read">
2326     <wsdl:input message="readSoapReq"/>
2327     <wsdl:output message="readSoapRes"/>
2328   </wsdl:operation>
2329   <wsdl:operation name="write">
2330     <wsdl:input message="writeSoapReq"/>
2331     <wsdl:output message="writeSoapRes"/>
2332   </wsdl:operation>
2333   <wsdl:operation name="invoke">
2334     <wsdl:input message="invokeSoapReq"/>
2335     <wsdl:output message="invokeSoapRes"/>
2336   </wsdl:operation>
2337 </wsdl:portType>
2338 <wsdl:binding name="oBIXSoapBinding" type="oBIXSoapPort">
2339   <soap:binding style="document"
2340     transport="http://schemas.xmlsoap.org/soap/http"/>
2341   <wsdl:operation name="read">
2342     <soap:operation soapAction="http://obix.org/ns/wsdl/1.0/read"
2343       style="document"/>
2344     <wsdl:input>
2345       <soap:body use="literal"/>
2346     </wsdl:input>
2347     <wsdl:output>
2348       <soap:body use="literal"/>
2349     </wsdl:output>
2350   </wsdl:operation>
2351   <wsdl:operation name="write">
2352     <soap:operation soapAction="http://obix.org/ns/wsdl/1.0/write"
2353       style="document"/>
2354     <wsdl:input>
2355       <soap:body use="literal"/>
2356     </wsdl:input>
2357     <wsdl:output>
2358       <soap:body use="literal"/>
2359     </wsdl:output>
2360   </wsdl:operation>
2361   <wsdl:operation name="invoke">
2362     <soap:operation soapAction="http://obix.org/ns/wsdl/1.0/invoke"
2363       style="document"/>
2364     <wsdl:input>
2365       <soap:body use="literal"/>
2366     </wsdl:input>
2367     <wsdl:output>
2368       <soap:body use="literal"/>
2369     </wsdl:output>
2370   </wsdl:operation>
2371 </wsdl:binding>
2372 </wsdl:definitions>

```

2373

Appendix A. Revision History

Rev	Date	By Whom	What
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 Hanson	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
wd-0.12.2	30 Jun 06	Aaron Hansen	OASIS document format compliance.

2374