oBIX Technical Deep-Dive

Qingtao Cao [qingtao.cao.au@gmail.com]

Last updated: 2015-2-19

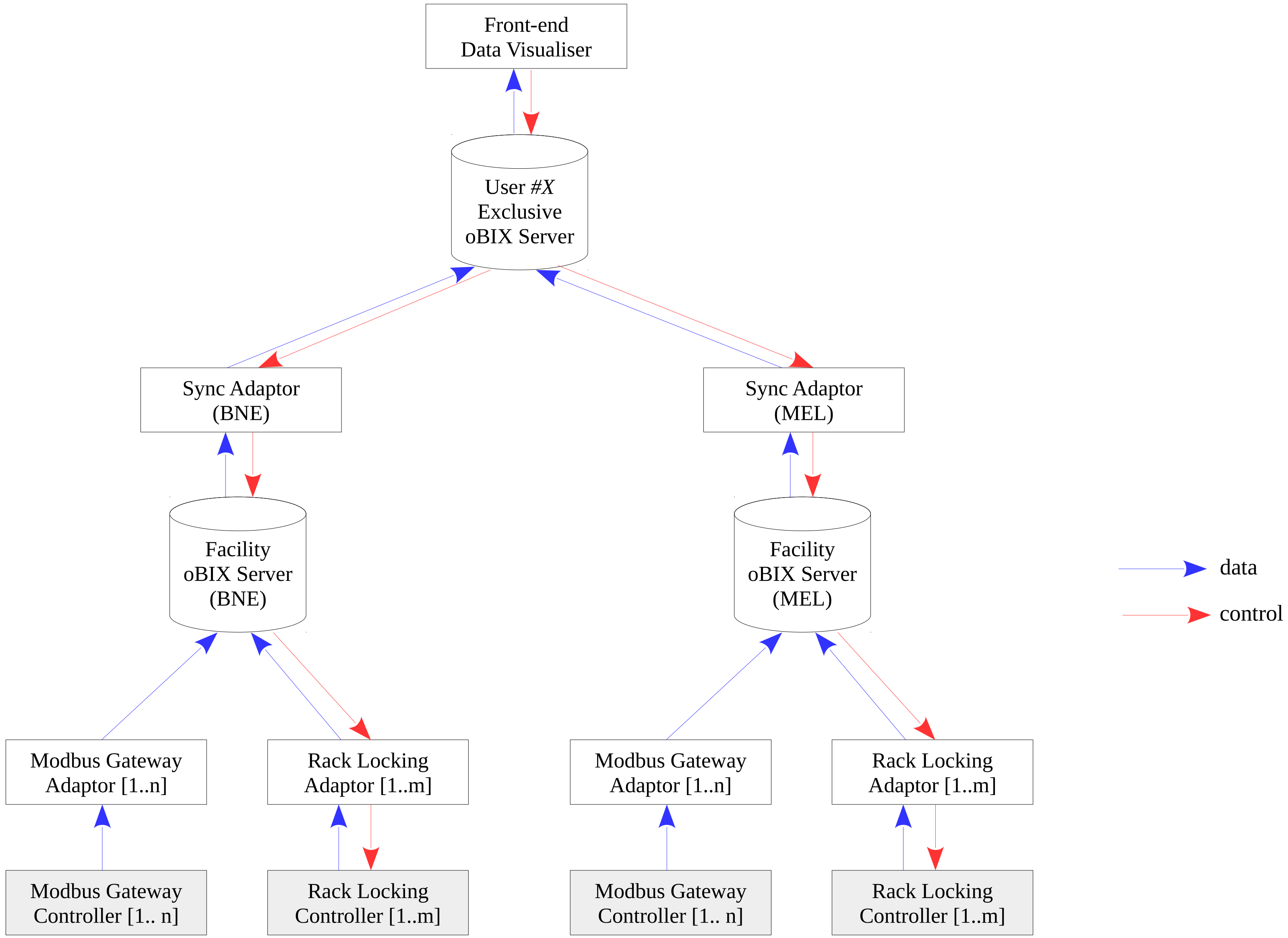
Table of Contents

Source Code:

https://github.com/Qingtao-Cao/obix.git

0. Bird's-eye-view of oBIX

0.1 oBIX as DCIM Backend



The oBIX server is fast, stable and scalable, it can be used to setup as backend for a Data Centre Information Management (or DCIM) software. One oBIX server can be setup at a high level for one particular user which is called Customer-Exclusive oBIX server and only contains information about the assets used by that specific user in various data centres. Such segregation of setting up different oBIX servers for different users not only promotes data security for users' information, but also further improve performance of oBIX servers by avoiding potential bottlenecks.

Each Customer-Exclusive oBIX server can have extra front-end facilities to assist in the visualisation of data before it is presented to relevant user in a more readable, more friendly format (compared with raw XML format). The content of each Customer-Exclusive oBIX server comes from a number of oBIX servers at a facility level, depending on which data centre the user has assets in.

For each data centre, there is one oBIX server established that contains all the data from all kinds of equipment within that data centre, such as modbus gateway controllers which connect modbus slave devices used for power monitoring, rack locking controllers responsible for lock management on racks, or ambient environment monitors that keep tracks of the temperature and humidity in data halls. Such oBIX server is referred to as the Facility oBIX server, which is user-neutral and has no record of which equipment is currently used by whom.

Meanwhile, an oBIX adaptor, named Sync Adaptor is responsible for extracting data for a single specific user from one Facility oBIX server and populating them onto relevant Customer-Exclusive oBIX server. Normally there is one Sync Adaptor deployed for a specific user in one specific data centre.

In essence an oBIX server can be regarded as a XML data warehouse and implements low-level mechanism to add, remove and access data in a multi-thread environment, it relies on oBIX clients to implement high-level policy about data manipulation and oBIX adaptors to store and update their data in the first place. No matter what different types of controlling system a data centre may have adopted, an oBIX adaptor can be developed for it, talking its language to get information out through its specific interfaces. Then that oBIX adaptor will reformat the data into XML contract as suggested by the oBIX specification and finally relay onto the central oBIX server. If needed, a hierarchy structure of oBIX servers at different levels can further be setup. Ultimately, users can gain a “single panel of glass” view of their assets in different data centres through the top level oBIX server.

0.2 oBIX Real-Time Power Monitoring Demo

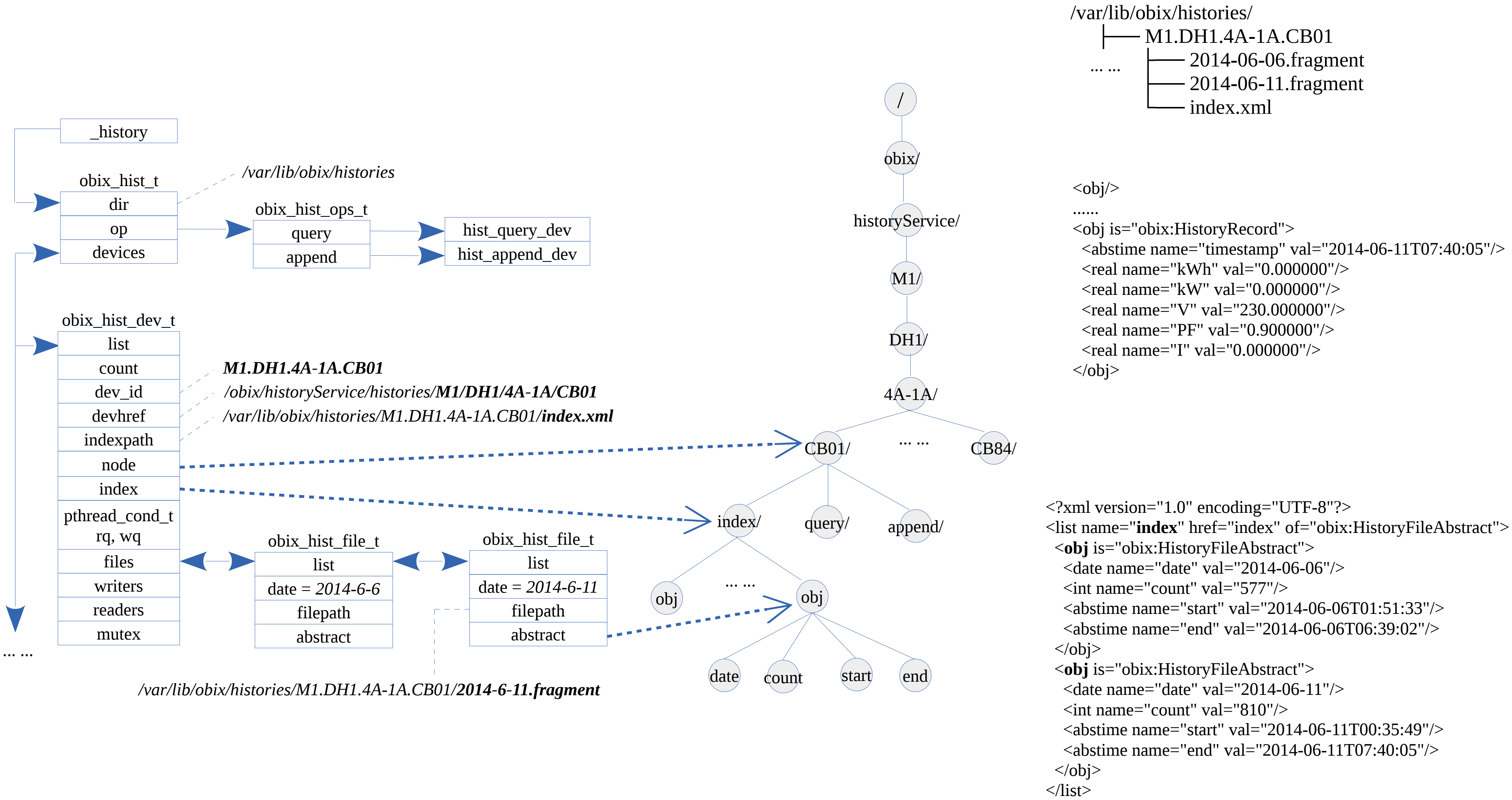
Key facts about a demonstration video of the real-time power monitoring using oBIX:

. 20 oBIX server threads are spawned and handling requests concurrently (the number of server threads is configurable);
. 5 mgate adaptors are running on the local machine along with the oBIX server, they are accessing 5 different modbus gateway controllers separately and can send 15 requests on different device contracts simultaneously;
. 3570 device contracts are registered and updated on the oBIX server asynchronously;
. 10 seconds will it take the oBIX server to refresh device contracts for hardware in local data centre;
. 30 seconds will it take the oBIX server to refresh device contracts for hardware in a different data centre at another city;
. snapshots of device contracts are backed up to hard-drive at configurable interval, such as 5 minutes;
. history records of device contracts are stored to hard-drive at configurable interval, such as 5 minutes;

Moreover, if oBIX server crashed or stopped, it can recovery the previous context from persistent device files and history facilities on the hard drive, no re-start of oBIX adaptors are required.

1. History Subsystem

1.1 Software Infrastructure



1.2 Filesystem Layout

The rightmost part of the above diagram shows the layout of history related folders and files on the hard-drive. oBIX clients can request the oBIX server to create a history facility for one particular device. On the hard-drive of the oBIX server, all such history facilities for different devices are organised under a particular folder. As illustrated in this example, under the folder of "/var/lib/obix/histories/" there is a history facility named "M1.DH1.4A-1A.CB01", which stands for the history facility for the Circuit Breaker #1 in a Branch Circuit Monitor named "4A-1A" in the Data Hall 1 of the M1 data centre.

This history facility contains one and only one index file in XML format and a number of raw history data files, each of which is a huge collection of XML objects representing history records. Since they don't have the required XML root elements, they are just XML fragments, that's why their file names are suffixed by "fragment". BTW, each history data file is named after the date when they were generated.

For XML objects representing history records, each of them consists of a timestamp element describing its generation time and a number of other elements as its payload. For example, the history record for a Circuit Breaker contains all the information about its power output at a particular moment, such as kWh, kW, Voltage, Power Factor and Current Intensity. As in this example, probably because no racks have been connected to this Circuit Breaker ever, both its kWh and Current Intensity values are zero.

The index file of a history facility contains abstract objects for all its fragment files. As shown in this example, they are for the raw history data files generated on 2014-6-6 and 2014-6-11 respectively. Aside from the generation date, the abstract object also shows the total number of records in a fragment file, the start and the end timestamp of the very first and the very last record in that file respectively.

Whenever a new history record is added to a history facility, it is actually appended to the very end of the ***latest*** fragment file with relevant abstract object updated accordingly. For instance, at least the count number is increased by 1 and the end timestamp is set equal to that of the newly appended history record.

BTW, it's worthwhile to mention that the history subsystem of the oBIX server enforces strict control on history records so that they are in strict timestamp ascending order, which prohibits oBIX clients from appending a history record with timestamp older than or equal to that of the last record in the latest fragment file.

Furthermore, if the new history record is on a brand new date, a new fragment file is created from scratch for that date with its abstract object inserted into the index file.

1.3 XML DOM Hierarchy

The middle part of the above diagram is a segment of the global DOM tree related with the history subsystem. The history facility for the Circuit Breaker 1 mentioned earlier has its own XML sub-tree, which has two child nodes named "query" and "append", representing the operations supported by this history facility, and the content of the index sub-tree is directly converted from relevant “index.xml” file on the hard-drive.

It's important to note that for the sake of performance and efficiency, only index files are loaded into the global DOM tree at the oBIX server's start-up. Whereas history fragment files may contain hundreds of thousands of records with several GB data, they are ***never*** loaded into the global DOM tree.

1.4 Software Descriptors

As shown in the leftmost part of the above diagram, an obix_hist_dev_t structure is created for every existing history facility on the hard-drive. They are all organised into a "devices" list in the obix_hist_t structure for the entire history subsystem. This way, the history subsystem is free from limitation on the number of history facilities. Similarly, for every fragment file an obix_hist_file_t structure is created and they are all organised into the "files" list in the obix_hist_dev_t descriptor for the relevant history facility. This way, there is no limitation on the number of fragment files contained in one single history facility.

The obix_hist_dev_t descriptor describes a history facility's name, its parent href in the global DOM tree and the absolute path of its index file on the hard-drive, while the obix_hist_file_t descriptor contains the information about a fragment file's generation date and its absolute path in the filesystem. Both software descriptors contain pointers pointing to relevant XML nodes in the global DOM tree, which are highlighted by bold, dotted and blue lines with arrow in the above diagram.

Whenever a new history record is appended, both software descriptors and relevant XML nodes are updated altogether, and the content of the index sub-tree is also converted into a XML document and saved into relevant index file on the hard-drive to prevent potential data loss.

1.5 Multi-thread Support

Different history facilities should allow concurrent accesses, that is, different users are permitted to append or query from different history facilities simultaneously. Moreover, although history facilities are not removable, new history facilities can be created while existing history facilities are accessed. Last but not least, multiple users may operate on one history facility asynchronously from each other.

Please refer to chapter 4 for how potential race conditions are properly addressed.

1.6 Performance of the Query Operation

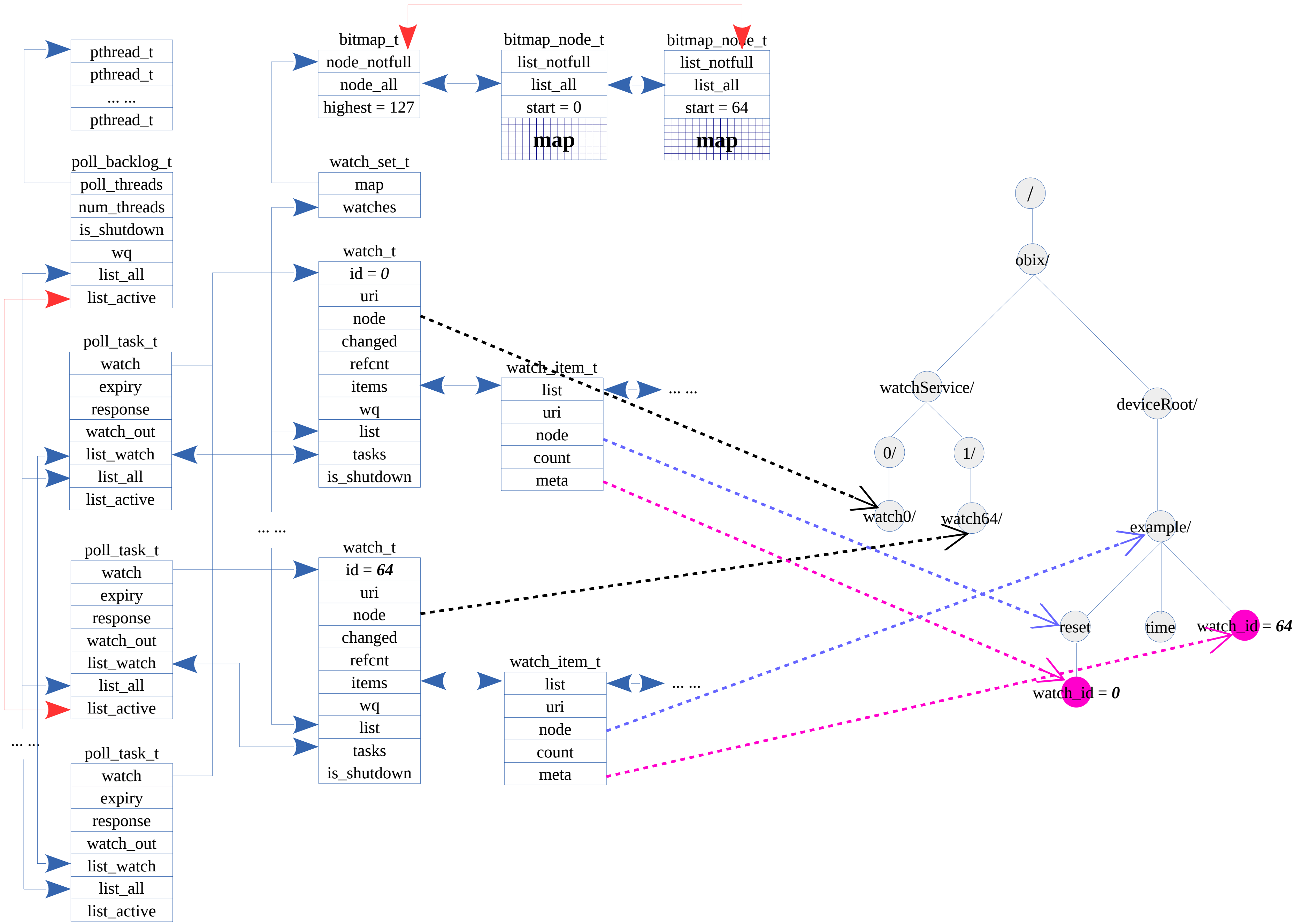
Having said earlier, a history facility may contain an enormous number of raw fragment files, each of which may further contain a vast number of records. So it's not unusual for one history facility to host several GB data, in this case the speed of the query operation could easily become a bottleneck of the entire history subsystem.

When an oBIX client is querying from a history facility, it specifies a limit on the number of records wanted and a timestamp range. The query operation will only return satisfactory records from the specified history facility. Suppose a number of fragment files are involved in one query request, thanks to the fact that records in a fragment file are in strict timestamp ascending order and the fragment files are organised by their generation date, normally speaking, only the records in the first and the last fragment files may need to be examined one after the other, whereas the content of all the rest fragment files in the middle could be directly returned back to relevant oBIX client.

Please refer to the source code for details about all those complicated mess of thoughts involved.

2. Watch Subsystem

2.1 Software Infrastructure



pthread_t
pthread_t
... ...
pthread_t
poll_backlog_t
poll_threads
num_threads
is_shutdown
wq
list_all
list_active
poll_task_t
watch
expiry
response
watch_out
list_watch
list_all
list_active
poll_task_t
watch
expiry
response
watch_out
list_watch
list_all
list_active
... ...
poll_task_t
watch
expiry
response
watch_out
list_watch
list_all
list_active
bitmap_t
node_notfull
node_all
highest = 127
bitmap_node_t
list_notfull
list_all
start = 0
map
bitmap_node_t
list_notfull
list_all
start = 64
map
watch_set_t
map
watches
watch_t
id = 0
uri
node
changed
refcnt
items
wq
list
tasks
is_shutdown
... ...
watch_t
id = 64
uri
node
changed
refcnt
items
wq
list
tasks
is_shutdown
watch_item_t
list
uri
node
count
meta
... ...
watch_item_t
list
uri
node
count
meta
... ...
/
obix/
watchService/
deviceRoot/
0/
1/
watch0/
watch64/
example/
reset
time
watch_id = 64
watch_id = 0

2.2 XML DOM Hierarchy

As shown by the right part of the above diagram, under the oBIX lobby there are two nodes for the watchService and the deviceRoot respectively. The watchService subtree contains XML contracts for every single watch objects created, each of which has a unique ID number and no more than 64 watch objects are grouped under their parent object. When there are tens of thousands of watch objects on one oBIX server and if they were simply organised as direct children of the watchService itself, it would take ages to traverse the whole subtree of it to look for one particular watch object with a specific ID number. That's why this 2-level hierarchy structure is adopted in order to promote performance.

The deviceRoot node hosts device contracts registered by oBIX clients. As shown in the diagram there is an “example” device registered under it with two children nodes of “reset” and “time”. Suppose the “reset” node is currently monitored by a watch object with ID 0 and the entire example device is watched upon by another watch object with ID 64, a special type of meta element is installed under each of them to store the ID information of the relevant watch object. This way, whenever a XML node in the global DOM tree are changed, the oBIX server has knowledge about whether it is being monitored by any watch object and whom should be notified of the change event.

It's also interesting to note what happens when two watch objects are monitoring the nodes in one same sub-tree at the same time. With the help of the meta elements installed in the global DOM tree, the oBIX server also has a clear idea of whether there is any watch object monitoring any ***ancestor*** nodes of the changed node. If this is the case, then the watch objects on the ancestors will be notified as well. For example, when the “reset” node is changed, both watch 0 and watch 64 are notified. However, only watch 64 is notified if other parts of the “example” device other than the “reset” node are changed.

BTW, if there are more than one watch objects monitoring one same XML node, multiple meta elements are created under it for each watch object respectively. And a meta element will be removed when relevant watch object no longer keeps an eye on that XML node.

2.3 Software Descriptors

The watch_set_t structure is the high-level descriptor of the entire watch subsystem, it has a “map” pointer pointing to a list of extensible bitmap nodes which provide ID numbers for new watch objects. When a watch object is deleted, its ID number is recycled properly.

Whenever a brand-new watch object is created, a watch_t descriptor is created and organised into the “watches” list in the watch_set_t descriptor. That's how the watch subsystem supports unlimited number of watches on the oBIX server. For every XML node monitored by one watch object, a watch_item_t descriptor is created accordingly and they are organised into the “items” list in relevant watch_t descriptor so that one watch object is able to monitor unlimited number of XML nodes in the global DOM tree.

Both the watch_t descriptor and the watch_item_t descriptor have pointers pointing to relevant XML nodes in the global DOM tree. In particular, the “node” pointer in the watch_t descriptor points to the watch object under the watchService, while the “node” pointer in the watch_item_t descriptor points to the monitored XML node. Furthermore, the watch_item_t descriptor also has a pointer named "meta" pointing to the special meta element installed under the monitored XML nodes in the global DOM tree.

2.4 Poll Threads and Poll Tasks

The poll_backlog_t structure is the high-level descriptor of relevant polling threads and all pending poll tasks. Whenever an oBIX client would like to use a watch object to monitor some XML nodes in the global DOM tree, it firstly requests to have a watch object created, then tells it which XML node or nodes to monitor. Finally the client forks a thread blocked to listen to the notification sent back from the watch object. Accordingly, a poll_task_t descriptor is created on the oBIX server that contains all the information about which watch object is being polled upon, the maximal waiting time of relevant poll task and a pointer to the FCGX response which will be loaded with the notification back to relevant oBIX client.

As a matter of fact, one poll_task_t structure can join as many as 3 different lists for different purposes.

In the first place, in order to support having one watch object shared among multiple oBIX clients, a poll_task_t descriptor is equipped with a “list_watch” field in order to join a “tasks” list in the relevant watch_t descriptor. When a watch object is being polled by multiple oBIX clients, relevant poll tasks are all organised in this queue. So whenever any watch item descriptor of this watch object detects any positive changes, all poll tasks in that queue are asserted so as to be further handled by a polling thread, so that in the very end, all oBIX clients sharing this watch object can get the same notification of changes.

In the second place, in order to enable the polling threads to take care of all polling tasks in an efficient manner, each poll_task_t descriptor can join two queues at the same time. In particular, upon creation a poll_task_t descriptor simply takes part in one universal “list_all” queue containing all poll tasks and its position in that queue is solely decided by its expiry date. This universal queue is organised according to the ascending order of the expiry date of each poll task, so that the polling threads are able to identify and handle those expired poll tasks just at the beginning of that queue effectively and efficiently.

In the third place, one poll_task_t descriptor will also join a separate, “list_active” queue right after relevant watch object is triggered by a change event, so that it is placed immediately on the radar of the polling threads, which will race to grab one poll task from the queue and handle it. It's worthwhile to mention that the polling threads always handle the active queue first, then the expired tasks at the beginning of the universal queue, then fall back asleep again until waken up by another change event or the first poll task(s) in the universal queue becomes expired.

Please further refer to chapter 4 about how synchronisation on the watchService lobby (among reading, adding and deletion of watches) and on one watch object (among readers and writers) are done with the help of tsync_t and refcnt_t structures embedded in watch descriptors.

2.5 Extensible, Recyclable Bitmaps

Previously a simple integer was used to count the ID number for watch objects, it would overflow in the future when a large number of watch objects were created and then deleted repeatedly, consequently, newly created watch objects may have same ID number as existing ones which is not acceptable at all since it is critical for a watch object to be assigned a unique ID number so as to be used by oBIX clients properly. To address this challenge a list of bitmap nodes are adopted, each of which covers 64 numbers from its starting number and they are organised in a “node_all” queue in the ascending order of their starting number.

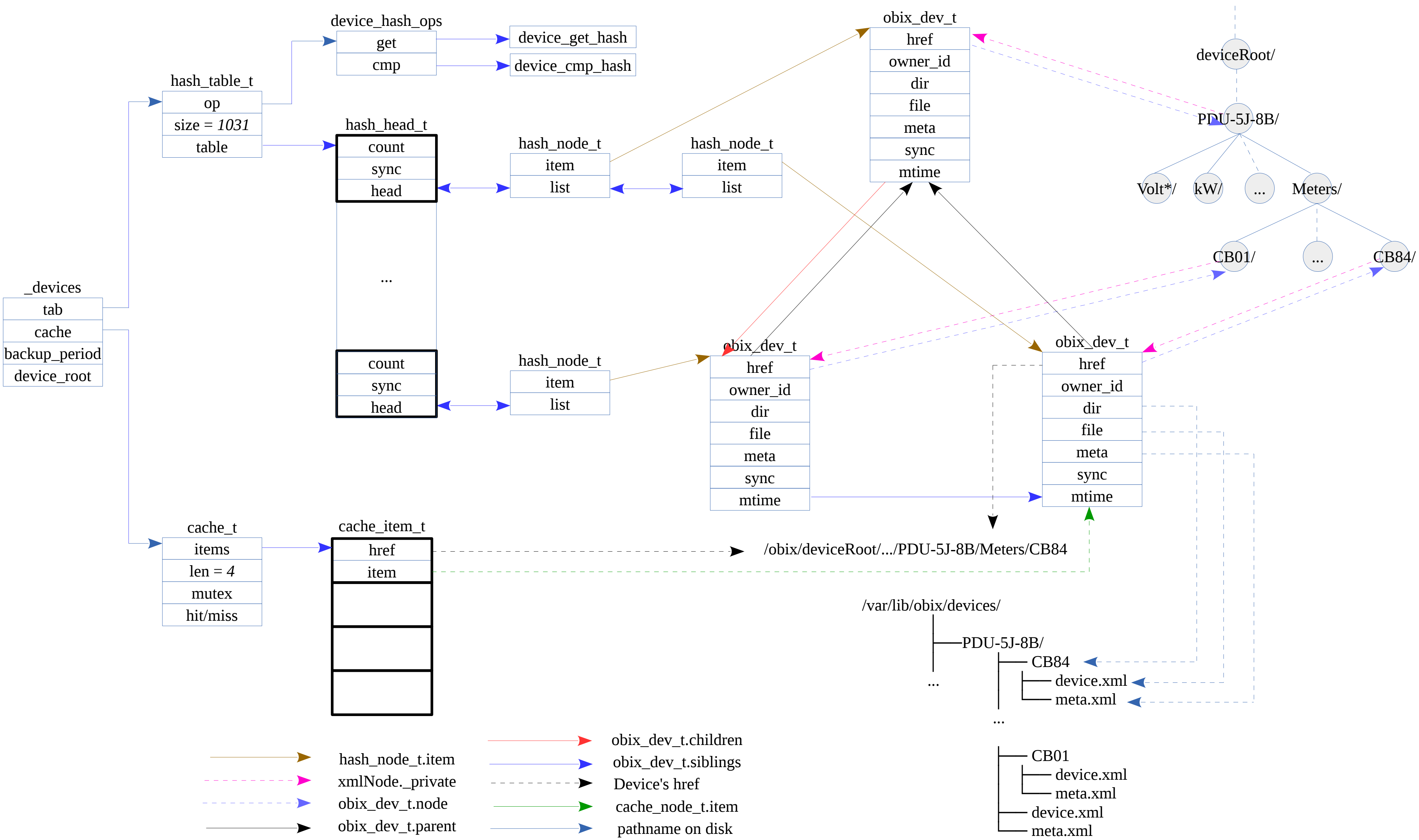
If a number is picked up for a new watch object, a bit in relevant bitmap node is asserted to 1, whereas whenever a watch object is deleted, its ID number is returned and relevant bit in relevant bitmap node is reset to 0.

If any bitmap node has more than one unused number, it also joins a separate “node_notfull” queue which is organised in strict ascending order of each bitmap node's starting number. This way, the ***smallest*** unused number is always returned and used.

Lastly, if a bitmap node has run out of available bits, it will be removed from the “node_notfull” queue. Of course, it will re-join this queue whenever any of its bits are returned and reset to 0 so as to be reused later.

3. Device Subsystem

3.1 Software Infrastructure



device_hash_ops
get
cmp
device_get_hash
device_cmp_hash
hash_table_t
op
size = 1031
table
hash_head_t
count
sync
head
...
count
sync
head
hash_node_t
item
list
hash_node_t
item
list
hash_node_t
item
list
obix_dev_t
href
owner_id
dir
file
meta
sync
mtime
obix_dev_t
href
owner_id
dir
file
meta
sync
mtime
obix_dev_t
href
owner_id
dir
file
meta
sync
mtime
deviceRoot/
PDU-5J-8B/
Volt*/
kW/
...
Meters/
CB01/
...
CB84/
_devices
tab
cache
backup_period
device_root
cache_t
items
len = 4
mutex
hit/miss
cache_item_t
href
item
/obix/deviceRoot/.../PDU-5J-8B/Meters/CB84
/var/lib/obix/devices/
PDU-5J-8B/
CB84
device.xml
meta.xml
...
...
CB01
device.xml
meta.xml
device.xml
meta.xml
hash_node_t.item
xmlNode._private
obix_dev_t.node
obix_dev_t.parent
obix_dev_t.children
obix_dev_t.siblings
Device's href
cache_node_t.item
pathname on disk

3.2 Software Descriptors

3.2.1 Device Descriptors

The device descriptors, obix_dev_t, sit at the heart of the hierarchy of the Device subsystem, acting as entry points for device contracts registered by oBIX clients. A device descriptor is established when a device contract is registered (signed up) and destroyed when the device contract is unregistered (signed off). It contains the meta data of the device contract such as the absolute href, ownership information, path names of relevant files on the hard drive and the timestamp of previous backup etc.

A device descriptor hosts a pointer to the root node of the subtree of relevant device contract in the XML DOM tree, meanwhile, ***every*** single node in that subtree uses their "_private" pointers (designed to store application specific data in libxml2) pointing back to their device descriptor so that whenever a node is accessed, relevant device descriptor is identified and then used to enforce synchronisation with other accesses on the same device contract.

With the help of device descriptors, getting the absolute href of a given node in the XML DOM tree has become really easy and fast. Unlike the old method used in the xmldb_node_path() to traverse from the given node all the way upward to the root node of the entire XML DOM tree (which is a performance-killer and non thread-safe), the search operation now only takes place within the boundary of relevant device contract.

Members of a device descriptor are further illustrated in the following sections.

3.2.2 Hash Table

As said above, the Device subsystem is the entry point of device contracts in the XML DOM tree, whenever a XML node with a particular href is accessed, its device descriptor is fetched from the Device subsystem so as to synchronise with other accesses to the same device in a multi-thread environment. In order to expedite searching for a specific device descriptor for a given href, descriptors are organised in a hash table based on the absolute href of their accompanied hrefs.

The hash table is created at the start-up of the Device subsystem before any device descriptor is ever created. The hash table contains an "op" pointer pointing to the "user" specific methods to compute a hash value from an identity and to compare whether two identities are the same. The "user" here refers to the user of the hash table, in this case, the Device subsystem, which provides its own functions to compute hash value from href strings and to compare whether the given href string is identical to the one in the device descriptor pointed to by a hash node.

The core part of the hash table is an array of hash_head_t structures, each of which leads a queue of hash_node_t structures that further refers to user-specific structures that yield the same hash value. It's interesting to mention that the size of the hash table is the smallest prime equal to or bigger than the value specified by relevant configuration option, the administrator is highly encouraged to tune according to the overall number of device contracts likely registered on the server in order to strike a better balance between time and space.

Note that the type of "item" pointers is (void *), with the help of it and the separated "op" function table, hash table APIs are neutral to its users. Better still, users can optimise their hash functions according to their specific use case. For example, two href strings are compared in a ***reversed*** manner, that is, from tail to head instead of the normal way around (via strcmp()) since hrefs tend to carry the same prefix of the Device lobby href and thus they are more likely to be distinct at tails.

There are a lot of discussions and comparison about hash functions suitable for strings, the one adopted by the Device subsystem is the hash_bkdr() function slightly modified to ignore the possible trailing slash at the tail of a href string.

The Debug version of the oBIX server supports a special URI to dump the entire content of the hash table, for example:

```
$ curl localhost/obix-dev-dump
  ......
  <list name="Queue 1018" of="obix:uri" len="1">
    <uri val="/obix/deviceRoot/M1/DH1/4I-10A/Meters/CB51/"/>
  </list>
  <list name="Queue 1019" of="obix:uri" len="1">
    <uri val="/obix/deviceRoot/M1/DH1/4I-10A/Meters/CB52/"/>
  </list>
  <list name="Queue 1020" of="obix:uri" len="1">
    <uri val="/obix/deviceRoot/M1/DH1/4I-10A/Meters/CB53/"/>
  </list>
  <list name="Queue 1021" of="obix:uri" len="2">
    <uri val="/obix/deviceRoot/M1/DH1/4C-5A/Meters/CB10/"/>
    <uri val="/obix/deviceRoot/M1/DH1/4I-10A/Meters/CB54/"/>
  </list>
  <list name="Queue 1022" of="obix:uri" len="2">
    <uri val="/obix/deviceRoot/M1/DH1/4C-5A/Meters/CB11/"/>
    <uri val="/obix/deviceRoot/M1/DH1/4I-10A/Meters/CB55/"/>
  </list>
  <list name="Queue 1023" of="obix:uri" len="2">
    <uri val="/obix/deviceRoot/M1/DH1/4C-5A/Meters/CB12/"/>
    <uri val="/obix/deviceRoot/M1/DH1/4I-10A/Meters/CB56/"/>
  </list>
  <list name="Queue 1024" of="obix:uri" len="2">
    <uri val="/obix/deviceRoot/M1/DH1/4C-5A/Meters/CB13/"/>
    <uri val="/obix/deviceRoot/M1/DH1/4I-10A/Meters/CB57/"/>
  </list>
  <list name="Queue 1025" of="obix:uri" len="3">
    <uri val="/obix/deviceRoot/M1/DH1/4C-5A/Meters/CB14/"/>
    <uri val="/obix/deviceRoot/M1/DH1/4I-10A/Meters/CB58/"/>
    <uri val="/obix/deviceRoot/M1/DH1/4I-12A/Meters/CB80/"/>
  </list>
......
```

Which basically enumerates hrefs of device descriptors in all conflict queues in the hash table.

3.2.3 Cache Facility

In order to manipulate the “locality principle” a cache is built on top of the hash table. As illustrated in the diagram, a cache node, or cache_item_t, hosts references to the href of a device contract and its descriptor. Each time before the hash table is searched the cache is traversed in the hope to shortcut the effort to find relevant device descriptor from the given href.

On a cache hit, the cache items won't be re-sorted and this makes it necessary to have the entire cached searched through each time. On a cache miss, the hash table is fallen back on and all cache items except the last one are moved one position downward so as to make room for a new cache node that is always “inserted” at the first cache slot.

Whenever a device contract is signed off from the server, its device descriptor is destroyed along with its subtree, so is its hash node. But any cache item referring to it, if exists in the first place, is simply nullified since href strings are not duplicated in cache nodes for sake of performance and efficiency.

Other consideration taken into the design of the cache facility include the size of it and whether it should cache href of sub nodes of a device contract (note they all refer to the same device descriptor). The larger the cache, the more significant amount of time is spent and wasted in cache maintenance and operations so it should be kept relatively small (currently 4 as specified in the server's configuration file). Given that oBIX clients can repeatedly update sub nodes in thousands of device contracts they managed, it's pointless to cache up subnode's href especially when the cache size is small.

The Debug version of the oBIX server supports a special URI to dump contents of all cache nodes, for instance:

```
$ curl localhost/obix-dev-cache-dump
<?xml version="1.0"?>
<obj name="Device Cache" hit="6917488" miss="8279381">
  <list name="Cache slots" of="obix:uri">
    <uri val="/obix/deviceRoot/M1/DH1/4A-2A/Meters/CB34/"/>
    <uri val="/obix/deviceRoot/M1/DH1/4I-11A/Meters/CB77/"/>
    <uri val="/obix/deviceRoot/M1/DH1/4A-2A/Meters/CB33/"/>
    <uri val="/obix/deviceRoot/M1/DH1/4G-8A/Meters/CB84/"/>
  </list>
</obj>
```

Note the counter of cache miss, it will explode if the cache size is increased.

3.3 Persistent Device

With the availability of device descriptors the "Persistent Devices” feature is now a low-hanging fruit to pick up. Device persistent files on the hard drive are created and deleted along with devices' signed up and signed off respectively, and written into hard drive periodically (further discussions see below). If the oBIX server crashed unexpectedly and re-started, it would recover its previous context by re-starting from existing persistent files to restore relevant device descriptors and is able to keep on receiving and processing requests from clients, who won't have to care about and aren't involved with oBIX server recovery at all.

The path names of a device's persistent files are stored in its descriptor at creation and manipulated by the Device subsystem throughout its life cycle.

3.3.1 Organisation of Persistent Files

Each device contract has its own folder on the hard-drive containing XML documents of its definition and the meta information, plus all sub-folders of its children device contracts. At server starts-up, the parent device contract is loaded before that of any of its children, so that the “mount point” in the parent device contract becomes available before any children device is inserted underneath.

Take a parent device, 4A-1A, for example, the layout of its folder on the hard-drive looks like below:

```
$ ls 4A-1A/
CB01  CB05  CB09  CB13  CB17  CB21  CB25  CB29  CB33  CB37  CB41  CB45  CB49  CB53  CB57  CB61  CB65  CB69  CB73  CB77  CB81  device.xml
CB02  CB06  CB10  CB14  CB18  CB22  CB26  CB30  CB34  CB38  CB42  CB46  CB50  CB54  CB58  CB62  CB66  CB70  CB74  CB78  CB82  meta.xml
CB03  CB07  CB11  CB15  CB19  CB23  CB27  CB31  CB35  CB39  CB43  CB47  CB51  CB55  CB59  CB63  CB67  CB71  CB75  CB79  CB83
CB04  CB08  CB12  CB16  CB20  CB24  CB28  CB32  CB36  CB40  CB44  CB48  CB52  CB56  CB60  CB64  CB68  CB72  CB76  CB80  CB84

$ ls 4A-1A/CB10/
device.xml  meta.xml
```

The “*device.xml*” file saves its latest (not in real-time manner) snapshot, for instance:

```
$ cat 4A-1A/device.xml
<?xml version="1.0" encoding="UTF-8"?><obj name="4A-1A" href="4A-1A" is="nextdc:veris-bcm">
  <int name="SlaveID" href="SlaveID" val="1"/>
  <int name="SerialNumber" href="SerialNumber" val="0x4e3458a0"/>
  <int name="Firmware" href="Firmware" val="0x03ed03f4"/>
  <int name="Model" href="Model" val="15172"/>
  <int name="CTConfig" href="CTConfig" val="2"/>
  <str name="Location" href="Location" val=" Panel #1 "/>
  <real name="ACFreq" href="ACFreq" val="50.000000"/>
  <real name="VoltL-N" href="VoltL-N" val="230.000000"/>
  <real name="VoltL-L" href="VoltL-L" val="415.000000"/>
  <real name="VoltA" href="VoltA" val="230.000000"/>
  <real name="VoltB" href="VoltB" val="230.000000"/>
  <real name="VoltC" href="VoltC" val="230.000000"/>
```

```
  <real name="kWh" href="kWh" displayName="Total kWh for 3 phases" val="0.000000"/>
  <real name="kW" href="kW" val="8.960876"/>
  <real name="CurrentAverage" href="CurrentAverage" val="0.000000"/>
  <real name="TotalCurrent" href="TotalCurrent" val="43.289261"/>
  <abstime name="LastUpdated" href="LastUpdated" val="2015-01-12T01:24:31Z"/>
  <bool name="Online" href="OnLine" val="true"/>
  <list name="Meters" href="Meters" of="nextdc:Meter"/>
</obj>
```

The device persistent file of a children device is similar to its history record:

```
$ cat 4A-1A/CB10/device.xml
<?xml version="1.0" encoding="UTF-8"?>
<obj name="CB10" href="CB10" is="nextdc:veris-meter">
  <real name="kWh" href="kWh" val="0.000000"/>
  <real name="kW" href="kW" val="1.449378"/>
  <real name="V" href="V" val="230.000000"/>
  <real name="PF" href="PF" val="0.900000"/>
  <real name="I" href="I" val="7.001825"/>
</obj>
```

It's worthwhile to mention that the parent device's persistent file does ***not*** contain any of its children devices. As illustrated in above example, the “Meters/” list in the parent's contract is empty and its children devices are in fact in each of their own device files. Obviously, if children devices were also present in their parent device's persistent files, they would be loaded twice at start-up which is clearly wrong and must be avoided.

Furthermore, the “*meta.xml*” file contains a device contract's meta information such as its absolute href and the ownership information which are not available in the device contract:

```
$ cat 4A-1A/meta.xml
<?xml version="1.0" encoding="UTF-8"?>
<obj of="nextdc:device-meta">
   <str name="owner_id" val="UNDEFINED:UNDEFINED"/>
   <uri val="/obix/deviceRoot/M1/DH1/4A-1A/"/>
</obj>

$ cat 4A-1A/CB10/meta.xml
<?xml version="1.0" encoding="UTF-8"?>
<obj of="nextdc:device-meta">
   <str name="owner_id" val="UNDEFINED:UNDEFINED"/>
```

```
  <uri val="/obix/deviceRoot/M1/DH1/4A-1A/Meters/CB10/"/>
</obj>
```

In particular, the “val” attribute of the <uri/> subnode tells oBIX server where to sign up relevant device contract when its persistent file is loaded.

3.3.2 When Persistent Files Should Be Updated?

It's a food-for-thought to make a sensible decision when a device contract should be backed up into its relevant file on the hard-drive.

In theory, the device persistent files can be updated whenever the device contract has been changed, for example, when the “val” attribute of a subnode is changed, (however, pay attention that the insertion or deletion of a subtree of a child device should not trigger the sync operation of the parent device, since children devices have their own persistent files) but in practice, a sync up threshold is imposed to limit the rate of disk I/O for sake of efficiency and performance.

In the first place, writing into disk files for every write attempt is redundant, especially for clients who constantly have their device contracts updated such as the MGATE adaptor. Come to think of it, the major purpose of device persistent files is to help the server recover from a catastrophic event, therefore the availability of files on the hard-drive outweighs whether their content are up-to-date or not.

In the second place, an overwhelming amount of disk I/O will definitely slow down the server and the situation deteriorates significantly if there are not enough number of server threads. Once the backlog queue of the FCGX listening socket has been filled up, newly arrived requests are simply rejected resulting in client side receiving HTTP 503 “Service Not Available” type of error.

So it's wise to enforce certain amount of latency between consecutive write attempts into hard-drive. To this end, a time_t structure, “mtime”, is introduced in device descriptors recording when the previous write operation took place. Only when the current write operation happens sufficiently later than the previous one would the persistent file on hard-drive be updated. The “dev-backup-period” configuration option specifies the minimal interval between two consecutive write operations.

Now it seems the answer has been found. However, there is more to think about. The problem arises when backing up a composite device contract that consists of a number of sub nodes and each of them is updated independently from another. If the device file and the “mtime” timestamp is updated along with the change of any subnode, changes to other sub nodes won't be saved into disks until the specified interval has elapsed.

The problem lies in that the oBIX server has no idea about the definition of a device contract, consequently it has no knowledge at all when the updates to various sub nodes of a device contract have been finished, that's when a write-through to disk should be scheduled, if desirable.

The solution to this problem comes to the usage of the batch object. As long as client side applications always make use of a batch object to have various sub nodes in a device contract updated, the oBIX server will have an idea when the update to the whole device contract completes, that is, when all sub commands in a batch object are properly handled.

To this end, oBIX adaptors are strongly recommended to manipulate batch objects to update their device contracts, even if only one subnode needs to be changed.

3.4 Synchronisation of Device Contracts

In order to support concurrent accesses to device contracts, POSIX pthread mutex and conditionals are used to implement critical regions for them. Relevant source code (tsync.c) also provides a number of APIs that can be easily manipulated by upper level software such as various subsystems on the oBIX server or the hash table.

Following table summarises how a device contract in the global DOM tree is protected from race conditions:

Access	Synchronisation Method
Read	Enter the “read regions” of the target device and its children devices (*)
Write	Enter the “write region” of the target device
SignUp	Enter the “write region” of the parent device
SignOff	Enter the “write region” of the parent device

The “read region” and “write region” are implemented by respective pthread conditionals and counters on the number of readers and writers. Multiple readers are allowed but exclusive to any writer on the same device, writers are exclusive to each other and ensured won't get starved by readers if they kept coming in. If any reader or writer should be blocked, relevant thread would be suspended on pthread conditionals instead of spinning on the mutex.

Please refer to the next chapter for more information on this topic.

(*) The parent device's subtree contain those of its children devices, transitioning from the parent's subtree into a children's subtree requires coming across the “read region” of the parent device into the “read region” of relevant children device, and returning back to the parent's when the children device has been read. The “_private” pointers of the xmlNode structures are compared to decide whether such transition is required. To this end, every single node in the subtree of a device contract has their “_private” pointer initialised pointing to their device descriptor when signed up.

4. Multi-thread Support

Before oBIX 2.0 release the oBIX server is a single-thread FCGX application which can't accept and handle more than one FCGX request at a time. Consequently, established FCGX requests are pending in the backlog queue of the listening socket of the oBIX server and are handled in a FIFO manner. If the oBIX server can't consume established requests fast enough, for example, when it was time-consuming to handle one request that triggers a noticeable disk operations to write back device contracts to their persistent files, or when FCGX requests simply pouring in quickly, incoming FCGX requests are simply rejected when the backlog queue of the listening socket is full resulting in client side getting HTTP 503 “Service Not Available” kind of error, which may further be exploited by hackers to launch DOS attack to the oBIX server.

Obviously the multi-thread support is the key to tackle with the above situation and to reinforce both the performance and the scalability of the oBIX sever, especially when dozens of or even hundreds of oBIX clients or adaptors are communicating with one oBIX server simultaneously.

4.1 Classic Race Conditions

One of the most classic hot bed for race conditions in a multi-thread environment is to split the find and access operations to a data structure in two steps. For example, the obix_server_read() API used to be implemented in the following pseudo code:

```
obix_server_read(const xmlChar *uri, ...)
        xmlNode *node, *copy;
         if ((node = xml_get_node(uri)) != NULL) {
                copy = xml_copy_node(node);
        }
```

First, the global XML DOM tree is searched for a particular node according to the given URI parameter, then it's copied so as to return to relevant requester as a response. Such behaviour to split the “get” and “copy” operations on a xml node works fine in a single-thread environment where no other threads existed at all to race against the current thread accessing it, however, it will bring about chaos in a multi-thread environment. Consider the following scenarios:

1. What if the target node, once its address is just returned to this thread, is deleted by others before its address is further referenced by this thread?
2. What if while searching for the desirable node it is being deleted from its parent, will the search operation fail or succeed?
3. What if while copying the desirable node, it is being updated?

Scenario #1 will result in segmentation faults without a doubt, scenarios #2 and #3 are both race conditions with unpredictable results. Basically enabling the multi-thread support of the oBIX server involves identifying and eliminating all potential race conditions.

4.2 Protection and Synchronisation on XML Node

The oBIX server can be regarded as a huge XML database with XML nodes organised in its global DOM tree representing different identities such as device contracts, history facilities or watch objects. Some XML nodes are static and installed at server start-up while others are dynamically created, changed and deleted. In essence the multi-thread support of oBIX server requires providing synchronisation among accesses to those volatile XML nodes, which can be categorised as either readers or writers. Readers behave in a read-only manner such as searching a subnode inside the given subtree or copying its content, whereas writers change its content (or its subtree) such as updating the val attribute of a subnode, adding or deleting a subnode from it etc.

Following principles have been adopted to synchronise among readers and writers of a resource:

1. Readers are inclusive with other readers;
2. Writers are mutually exclusive to each other, which means one and only one writer may exist at any time;
3. When the current writer exits, it will wake up ***one*** of pending writers, if they exist, before waking up ***all*** pending readers;
4. No more readers are allowed if there is any pending writers

The last two rules grant writers a higher priority than readers. In particular, rule #4 ensures pending writers won't get starved due to readers that may keep coming in like a rain that never stops.

Given that core data structures (or descriptors) in various subsystems throughout the oBIX server share similar requirement to act as portals to relevant xml nodes (or subtree) they are representing and exert access control and synchronisation among threads accessing them, APIs to enter and exit a “read region” or “write region” of a particular resource are abstracted and implemented in tsync.c

The tsync_t structure manipulates POSIX pthread mutex and conditionals and implements critical regions for its hosting data structure:

```
typedef struct tsync {
    int being_shutdown;
    int readers, writers;
    int running_readers, running_writers;
    pthread_cond_t rq, wq;
    pthread_cond_t swq;
    pthread_mutex_t mutex;
} tsync_t;
```

The being_shutdown is the flag indicating whether the hosting descriptor is being deleted and will block any further readers or writers from accessing it. The deletion thread uses this flag and the swq conditional to synchronise with existing readers and writers.

The readers and writers are counters for all readers and writers respectively, regardless of their status (running or blocked), whereas running_readers and running_writers are counters for active readers or writers that have successfully entered relevant critical region and are accessing the shared resource for the time being. Since writers are mutually exclusive to each other the running_writers can't be more than 1. New readers are blocked on rq conditional due to running or pending writers, and new writers are blocked on wq conditional because of any readers or writers.

With the help of these elements contained in a tsync_t structure, APIs to manipulate critical regions can be implemented as below

```
int tsync_writer_entry(tsync_t *sync)
{
    pthread_mutex_lock(&sync->mutex);

    if (sync->being_shutdown == 1) {
        pthread_mutex_unlock(&sync->mutex);
        return -1;
    }

    sync->writers++;

    while (sync->running_readers > 0 || sync->running_writers > 0) {
        pthread_cond_wait(&sync->wq, &sync->mutex);
```

```
    }

    sync->running_writers++;

    pthread_mutex_unlock(&sync->mutex);
    return 0;
}
```

Before a writer can enter the “write region” of relevant resource successfully, it needs to check if someone else has started deleting it and return a failure if this is the case. Then it needs to wait for the completion of any running readers or writers. It's worthwhile to mention that the check on relevant flags and wait on the wq conditional have to be placed in a loop so as to avoid ***spurious*** wake up. Also note that the overall writers counter is increased before sleeping while the running wrtiers count is only increased after entering the write region successfully.

```
void tsync_writer_exit(tsync_t *sync)
{
    pthread_mutex_lock(&sync->mutex);

    sync->writers--;
    sync->running_writers--;

    if (sync->writers > 0) {
        pthread_cond_signal(&sync->wq);
    } else if (sync->readers > 0) {
        pthread_cond_broadcast(&sync->rq);
    } else if (sync->being_shutdown == 1 && sync->readers == 0 && sync->writers == 0) {
        pthread_cond_signal(&sync->swq);
    }

    pthread_mutex_unlock(&sync->mutex);
}
```

When a writer exits the write region, it firstly decrease both counters related to writers. It will only wake up one pending writer, if exists, before wake up all pending readers. Lastly, if there is any pending deletion thread, it will only wake that up provided that there is neither pending writers nor readers.

```
int tsync_reader_entry(tsync_t *sync)
{
    pthread_mutex_lock(&sync->mutex);

    if (sync->being_shutdown == 1) {
```

```
        pthread_mutex_unlock(&sync->mutex);
        return -1;
    }

    sync->readers++;

    while (sync->writers > 0) {
        pthread_cond_wait(&sync->rq, &sync->mutex);
    }

    sync->running_readers++;

    pthread_mutex_unlock(&sync->mutex);
    return 0;
}
```

Readers APIs are implemented in a similar manner. Only note that since readers are inclusive to each other, they only need to wait for writers no matter whether they are running or being blocked, which further ensure no writers will get starved because of incoming readers.

```
void tsync_reader_exit(tsync_t *sync)
{
    pthread_mutex_lock(&sync->mutex);

    sync->readers--;
    sync->running_readers--;

    if (sync->running_readers == 0 && sync->writers > 0) {
        pthread_cond_signal(&sync->wq);
    }

    if (sync->being_shutdown == 1 && sync->readers == 0 && sync->writers == 0) {
        pthread_cond_signal(&sync->swq);
    }

    pthread_mutex_unlock(&sync->mutex);
}
```

The last reader will wake up any blocked writer, and only when there is no writer waiting will it wake up any pending deletion thread.

Lastly, a deletion thread will invoke the following function to synchronise with existing readers or writers and any other deletion thread on the same resource:

```
int tsync_shutdown_entry(tsync_t *sync)
{
    pthread_mutex_lock(&sync->mutex);
    if (sync->being_shutdown == 1) {
        pthread_mutex_unlock(&sync->mutex);
        return -1;
    }

    sync->being_shutdown = 1;

    while (sync->readers > 0 || sync->writers > 0) {
        pthread_cond_wait(&sync->swq, &sync->mutex);
    }

    pthread_mutex_unlock(&sync->mutex);
    return 0;
}
```

In particular, if the being_shutdown flag has been raised, some other thread must have started deleting the same resource, then have THAT thread delete it and exit directly. Before this function returns any existing readers and writers are ensured completed. Given that there will be no more readers or writers the deletion thread can safe get rid of relevant resource.

Last but not least, in some cases different critical regions of various descriptors need to be traversed when handling one request. For example, reading from lobby objects such as “/obix/deviceRoot/”, “/obix/watchService/” or “/obix/historyService/histories/”. Only when a XML node managed by a descriptor is read should its read region be entered so that other readers or writers on different objects can take place in parallel.

4.3 Protection and Synchronisation on Descriptors

As illustrated in above section, XML nodes rely on their accompanied core descriptors to enforce access control and synchronisation. Descriptors in various subsystems including obix_dev_t for a device contract, obix_hist_dev_t for a history facility and obix_watch_t for a watch object all manipulate a tsync_t structure to implement the critical regions for their accompanied XML nodes (or subtree).

However, another question surfaces: how to protect these descriptors from deletion?

Although history facilities don't support deletion, device contracts can be signed up or off on the fly and watch objects are created and deleted dynamically as well. Relevant descriptors are removed along with their counterpart in the XML global DOM tree. However, unfortunately, there may be other threads still holding a reference to a descriptor while it is being deleted. For example, the device_copy_uri() which copy a subtree of a particular URI from a device contract can be implemented in below pseudo code:

```
device_copy_uri(const xmlChar *href):
    obix_dev_t *dev;
    xmlNode *node, *copy = NULL;

    if (!(dev = device_search(href)) && !(dev = device_search_parent(href))) {
        return NULL;
    }

   /* How to prevent "dev" from being deleted here? */

    if (tsync_reader_entry(&dev->sync) == 0) {
        if ((node = __device_get_node_core(dev, href)) != NULL) {
            copy = __device_copy_node(dev, node, 0);
        }

        tsync_reader_exit(&dev->sync);
    }

    device_put(dev);
    return copy;
```

Firstly the hash table in the Device subsystem is searched for a suitable device descriptor that corresponds to a subtree matching the given href. If such device doesn't exist its parent device is fallen back on, which is the case when copying a subnode (or subtree) from a device contract. Then the current thread enters the “read region” of this device before it can safely go on to search for a child node inside the device contract and then make a copy of it. Lastly the current thread exits the “read region” of the device once done with it.

As long as the current thread has entered the “read region” of the device then it's ensured unable to be deleted (referring to the implementation of tsync_shutdown_entry() in previous section) until the reader exits. However, what if some other thread deleted the same device descriptor ***before*** this thread enters its “read region”?

To address this race condition, device descriptors host a “refcnt_t” structure along with the tsync_t structure to provide further synchronisation on device descriptors themselves. The refcnt_t structure can be regarded as a simplified tsync_t structure:

```
typedef struct refcnt {
    int count;
    pthread_mutex_t mutex;
    pthread_cond_t wq;
} refcnt_t;
```

It only contains one reference counter which is increased by one whenever a pointer to the hosting data structure is returned. Accordingly, users have the responsibility to decrease its reference counter whenever done with it, if it drops to zero, then any pending deletion thread would be finally waken up. The POSIX pthread mutex ensures the reference count is manipulated in a synchronised manner.

The refcnt.c file provides APIs such as refcnt_get() and refcnt_put() to increase and decrease the reference count in the given refcnt_t structure respectively, and refcnt_sync() to be invoked by deletion thread to wait for the completion of any outstanding users. These APIs are used by various subsystems on the oBIX server to protect core descriptors from being deleted unexpectedly. In the above example, device_put() simply invokes refcnt_put() on the embedded refcnt_t structure in a device descriptor:

```
static void device_put(obix_dev_t *dev)
{
    refcnt_put(&dev->refcnt);
}
```

Furthermore, in order to preciously increase the reference counter of device descriptor before returning its pointer, the hash table exports a “get” function hook so that the user of the hash table can provide a callback function for this purpose:

```
static hash_table_ops_t device_hash_ops = {
    .compute = device_compute_hash,
    .compare = device_compare_str,
    .get = device_get_wrapper
};

void *hash_search(hash_table_t *tab, const unsigned char *key)
    const void *item = NULL;
    hash_head_t *head;
    hash_node_t *node;

    head = &tab->table[tab->op->compute(key, tab->size)];

    if (tsync_reader_entry(&head->sync) < 0) {
        return NULL;
    }

    list_for_each_entry(node, &head->head, list) {
        if (tab->op->compare(key, node) == 1) {
            item = node->item;
            tab->op->get((void *)item);
            break;
```

```
        }
    }

    tsync_reader_exit(&head->sync);
    return (void *)item;
}
```

The get callback function is invoked before the address of a matching device descriptor is returned. As for the Device subsystem, relevant function simply utilises refcnt_get():

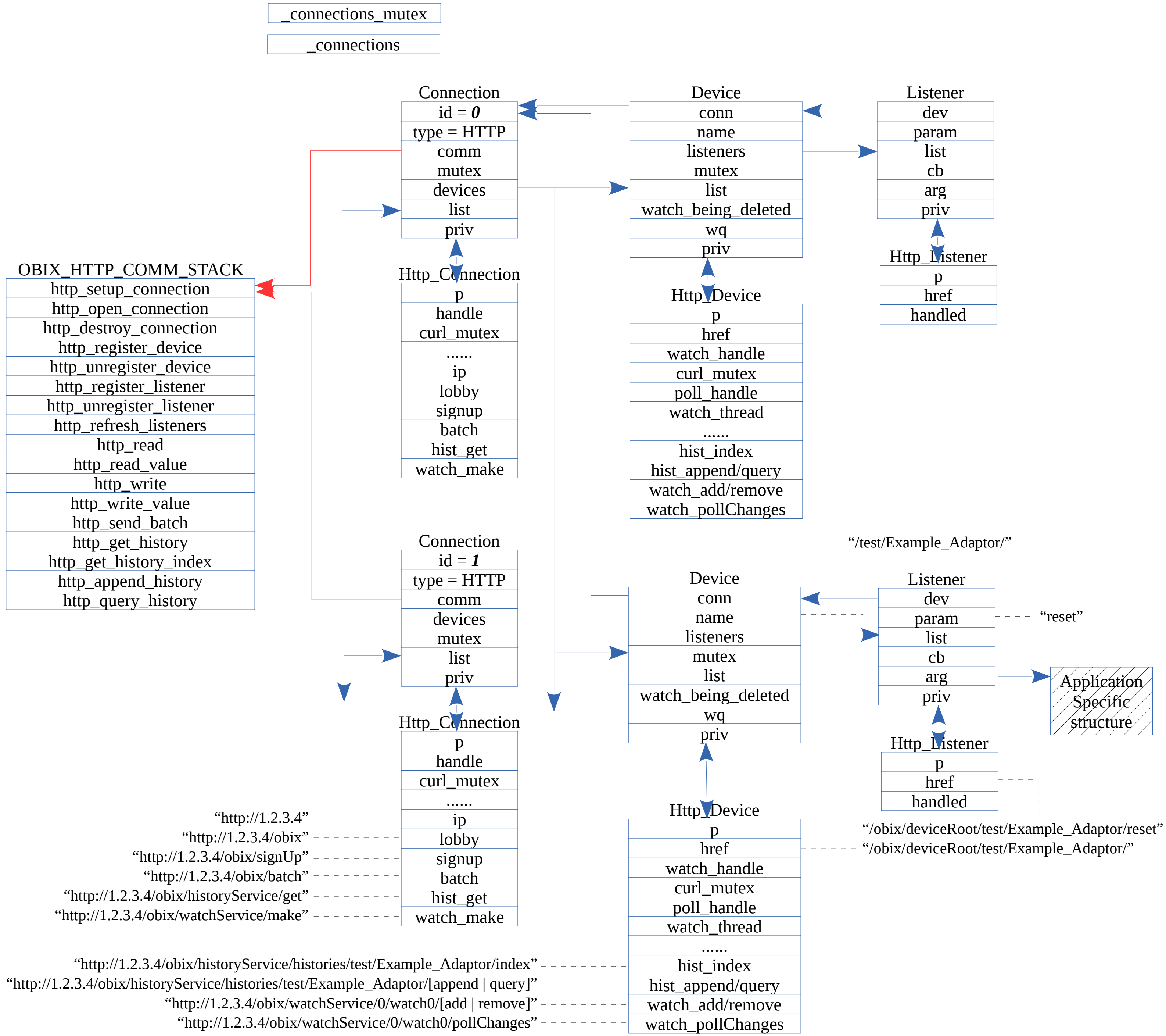
```
static void device_get_wrapper(void *dev)
{
    device_get((obix_dev_t *)dev);
}

static void device_get(obix_dev_t *dev)
{
    refcnt_get(&dev->refcnt);
}
```

Lastly, it's also worthwhile to mention that sometimes device_get() is needed explicitly, for instance, when it comes to the device descriptor of the root device of “/obix/deviceRoot/” which is not organised in the hash table, or when a device descriptor is referenced directly by the “_private” pointer of a XML node.

5. Client Side Core Structures

5.1 Software Infrastructure



_connections_mutex
_connections
Connection
id = 0
type = HTTP
comm
mutex
devices
list
priv
Http_Connection
p
handle
curl_mutex
......
ip
lobby
signup
batch
hist_get
watch_make
Device
conn
name
listeners
mutex
list
watch_being_deleted
wq
priv
Http_Device
p
href
watch_handle
curl_mutex
poll_handle
watch_thread
......
hist_index
hist_append/query
watch_add/remove
watch_pollChanges
Listener
dev
param
list
cb
arg
priv
Http_Listener
p
href
handled
OBIX_HTTP_COMM_STACK
http_setup_connection
http_open_connection
http_destroy_connection
http_register_device
http_unregister_device
http_register_listener
http_unregister_listener
http_refresh_listeners
http_read
http_read_value
http_write
http_write_value
http_send_batch
http_get_history
http_get_history_index
http_append_history
http_query_history
Connection
id = 1
type = HTTP
comm
devices
mutex
list
priv
Http_Connection
p
handle
curl_mutex
......
ip
lobby
signup
batch
hist_get
watch_make
"http://1.2.3.4"
"http://1.2.3.4/obix"
"http://1.2.3.4/obix/signUp"
"http://1.2.3.4/obix/batch"
"http://1.2.3.4/obix/historyService/get"
"http://1.2.3.4/obix/watchService/make"
"/test/Example_Adaptor/"
Device
conn
name
listeners
mutex
list
watch_being_deleted
wq
priv
Listener
dev
param
list
cb
arg
priv
"reset"
Application Specific structure
Http_Listener
p
href
handled
"/obix/deviceRoot/test/Example_Adaptor/reset"
"/obix/deviceRoot/test/Example_Adaptor/"
Http_Device
p
href
watch_handle
curl_mutex
poll_handle
watch_thread
......
hist_index
hist_append/query
watch_add/remove
watch_pollChanges
"http://1.2.3.4/obix/historyService/histories/test/Example_Adaptor/index"
"http://1.2.3.4/obix/historyService/histories/test/Example_Adaptor/[append | query]"
"http://1.2.3.4/obix/watchService/0/watch0/[add | remove]"
"http://1.2.3.4/obix/watchService/0/watch0/pollChanges"

5.2 Software Descriptors

The high-level oBIX software descriptors such as Connection, Device and Listener are low-level or binding neutral, which aim to describe a connection with an oBIX server, a device contract registered on it and a child node of that device monitored by a watch object on the server respectively. Meanwhile, they rely on low-level, HTTP counterparts such as Http_Connection, Http_Device and Http_Listener to host respective HTTP specific details on different identities.

5.2.1 Comm_Stack

The set of function pointers in the Comm_Stack structure provide an interface between high-level, oBIX APIs and low-level, binding specific implementation. The oBIX project implements only HTTP bindings for the time being and relevant methods are organised within an OBIX_HTTP_COMM_STACK structure. If SOAP bindings were to develop in the future, they should be arranged in a similar structure and a connection could selectively use one of supported bindings according to its relevant configuration settings.

Generally speaking, oBIX APIs manipulate high-level data structures and further invoke low-level, HTTP APIs to manipulate low-level data structures to fulfill particular operations. However, this is not necessarily always true. For example, obix_get_history_ts() simply invokes another high-level API of obix_query_history() with a special historyFilter contract to get timestamp strings of the first and last records from relevant history facility directly, without involving extra low-level operations.

5.2.2 Connection and Http_Connection

oBIX client applications can setup connections with more than one oBIX server, for instance, one oBIX Adaptor that needs to push hardware data to multiple oBIX servers simultaneously, or the “Synchronise Adaptor” which normally picks up particular information from a low-level oBIX server and relay them onward to another one at a higher level. To this end, Connections are organised into a global linked list to support no limitation on the number of them and establishment or deletion of connections at run-time, although oBIX clients practically setup their connections at start-up and use them throughout their life cycle.

Currently oBIX clients utilise a XML configuration file to specify details of connections with different oBIX servers, such as their unique ID number, IP address, type information etc, they are stored into connection related structures. Having said this, for the time being only HTTP binding is implemented and therefore all Connections' “comm” pointers point to the same OBIX_HTTP_COMM_STACK structure.

The Http_Connection structure describes the HTTP specific details of a connection, such as relevant oBIX server's IP address, URIs for all public facilities on that oBIX server and the particular CURL handle created and used exclusively for this connection. All URIs are prefixed by the server's IP address for the convenience of libcurl APIs. Please refer to the following section about CURL handle for pitfalls when manipulating them.

5.2.3 Device and Http_Device

The Device structure represents the high-level information of a device (or a XML object or an oBIX contract) registered to a particular oBIX server. Devices registered through one Connection are organised in its “devices” linked list, which is protected by a mutex to enable dynamic Device insertion and deletion. Each Device uses a “conn” pointer pointing back to the parent Connection.

Each Device has a unique name on its connection with an oBIX server as specified by relevant configuration settings. However, devices on different connections with the same server may happen to have duplicated names, in order to ensure devices are recognisable by their names on the server, their names are always prefixed by another "parent_href" configuration settings which specifies the root location of the registered device contract under the global device lobby of the oBIX server. In theory IT administrator must ensure the uniqueness of the "parent_href" settings of different oBIX clients across the whole server.

The Http_Device structure describes the HTTP specific details of a device, such as its href on relevant oBIX server, URIs for various facilities supported by this device, the worker thread dedicated for watch.PollChanges requests and CURL handles used for raising blocking and non-blocking requests respectively.

5.2.4 Listener and Http_Listener

The Listener structure is the client side equivalent to the server side watchItem structure, containing high-level information of a monitored child node of a particular Device, such as its relative href on the Device, the callback function invoked to claim changes and relevant parameter. Generally speaking the parameter can point to application specific data structure used by the callback function to handle changes on the monitored child node.

Generally speaking, multiple Listeners can be registered to monitor different child nodes of a Device independently, they are organised in the Device's "listeners" linked list, which is protected by a mutex to enable dynamic insertion and deletion. Each Listener uses a "dev" pointer pointing back to the parent Device. The Device's Listeners list is traversed once its worker thread receives a change notification (the watchOut contract to be exact, which may contain a list of updated nodes on the Device) from the oBIX server and those concerned callback functions are invoked one after the other.

It's worthwhile to note that the client side Device specific worker thread, the CURL handles related with watch relevant requests and the server side watch object are all created in a ***lazy*** mode, that is, they are established and destroyed along with the first Listener and the last Listener respectively. So the needed infrastructure is set up only when they are really needed and applications that never make use of them won't waste any memory at all.

Last but not least, a special use case is to set "param" argument equal to "/" which will have the entire device monitored instead of any child node, since double slashes are regarded as one when assembling the href of the monitored node therefore the href of the whole device is watched upon.

5.3 CURL Handles

Firstly and most importantly, CURL handles are ***not*** thread-safe therefore multi-thread applications must synchronise accesses on them. Each thread can either have its own exclusive CURL handle or contend for a mutex before accessing a shared one.

For sake of performance applications are encouraged to use multiple CURL handles when necessary, especially when it comes to blocking requests such as watch.PollChanges when requesters are likely to be blocked by libcurl API until a notification is returned back from the oBIX server, in which case each requester should independently manipulate their own CURL handle in order to receive each of their respective notifications in a timely manner.

To this end, a Http_Device structure hosts a "poll_handle" CURL handle exclusively used by its own "watch_thread" worker thread to listen for any changes occurred on the associated parent Device. Furthermore, a Http_Device structure also hosts another CURL handle named "watch_handle" so that other non-blocking watch requests can be raised in parallel with any blocking watch.PollChanges requests on the same Device.

Since Listeners are added and deleted at run-time, the Http_Device structure uses a mutex “curl_mutex” to serialise accesses to the “watch_handle”. Whereas the “poll_handle” is solely used by the worker thread so no such protection is required.

Lastly, many oBIX APIs accept the address of a user provided CURL handle as the first parameter. If it is NULL, then Connection specific or Device specific default CURL handle is fallen back on. Otherwise user applications must synchronise their usage on their own CURL handle.

The CURL handles used for different types of requests are summarised in below table:

<table>
<tr><th colspan="2">Request Type</th><th>CURL Handle Used</th></tr>
<tr><td colspan="2">GET</td><td>User defined or Connection default</td></tr>
<tr><td colspan="2">PUT</td><td>User defined or Connection default</td></tr>
<tr><td rowspan="10">POST</td><td>signup</td><td>Connection default</td></tr>
<tr><td>batch</td><td>User defined or Connection default</td></tr>
<tr><td>history.Get</td><td>User defined or Connection default</td></tr>
<tr><td>history.Append</td><td>User defined or Connection default</td></tr>
<tr><td>history.Query</td><td>User defined or Connection default</td></tr>
<tr><td>watch.Make</td><td>Device's “watch_handle”</td></tr>
<tr><td>watch.Add</td><td>Device's “watch_handle”</td></tr>
<tr><td>watch.Remove</td><td>Device's “watch_handle”</td></tr>
<tr><td>watch.PollRefresh</td><td>Device's “watch_handle”</td></tr>
<tr><td>watch.PollChanges</td><td>Device's “poll_handle”</td></tr>
<tr><td colspan="2">Listeners' callbacks</td><td>Device's “poll_handle” (*)</td></tr>
</table>

As we can see, the default CURL handle of the entire Connection is good for single-thread applications, whereas multi-thread applications must manipulate their own CURL handles to avoid racing against the Connection default one which will be a performance bottleneck, especially when it comes to time-consuming requests such as history.Query when massive amount of data may be transmitted over a slow network.

Note (*): once the Device specific worker thread has received the watchOut notification from oBIX server, it traverses the list of Listeners and invokes a Listener's callback function with its own CURL handle if it is concerned about current changes.

5.4 Race Condition on Listeners List

Attentions must be paid when accessing a Device's Listeners list, following threads are racing against each other on it:

1) application threads to create a new Listener (through the Device's “watch_handle”);
2) application threads to remove an existing Listener (through the Device's “watch_handle”);
3) the Device's specific worker thread polling for changes on the device (through the Device's “poll_handle”).

In the first place, a mutex needs to be held in scenario #1 and another mutex is also needed to ensure synchronised access to the watch_handle. The former mutex must be held until the communication with the oBIX server is finished even if it may take a relatively long time. Come to think of it, if a thread is blocked waiting for responses from a busy oBIX server or through a slow connection, there won't be extra benefit to have other threads raising similar requests.

In the second place, the Device's worker thread will release the mutex before invoking a Listener's callback function, which is supposed to access application specific structures and be time-consuming. As a result, the Listeners list could be modified during the invocation of a callback function, therefore the worker thread must restart from the beginning of the list and skip over already handled ones. To this end, the Http_Listener structure hosts an indicator about whether it has been handled by this round of execution of the worker thread.

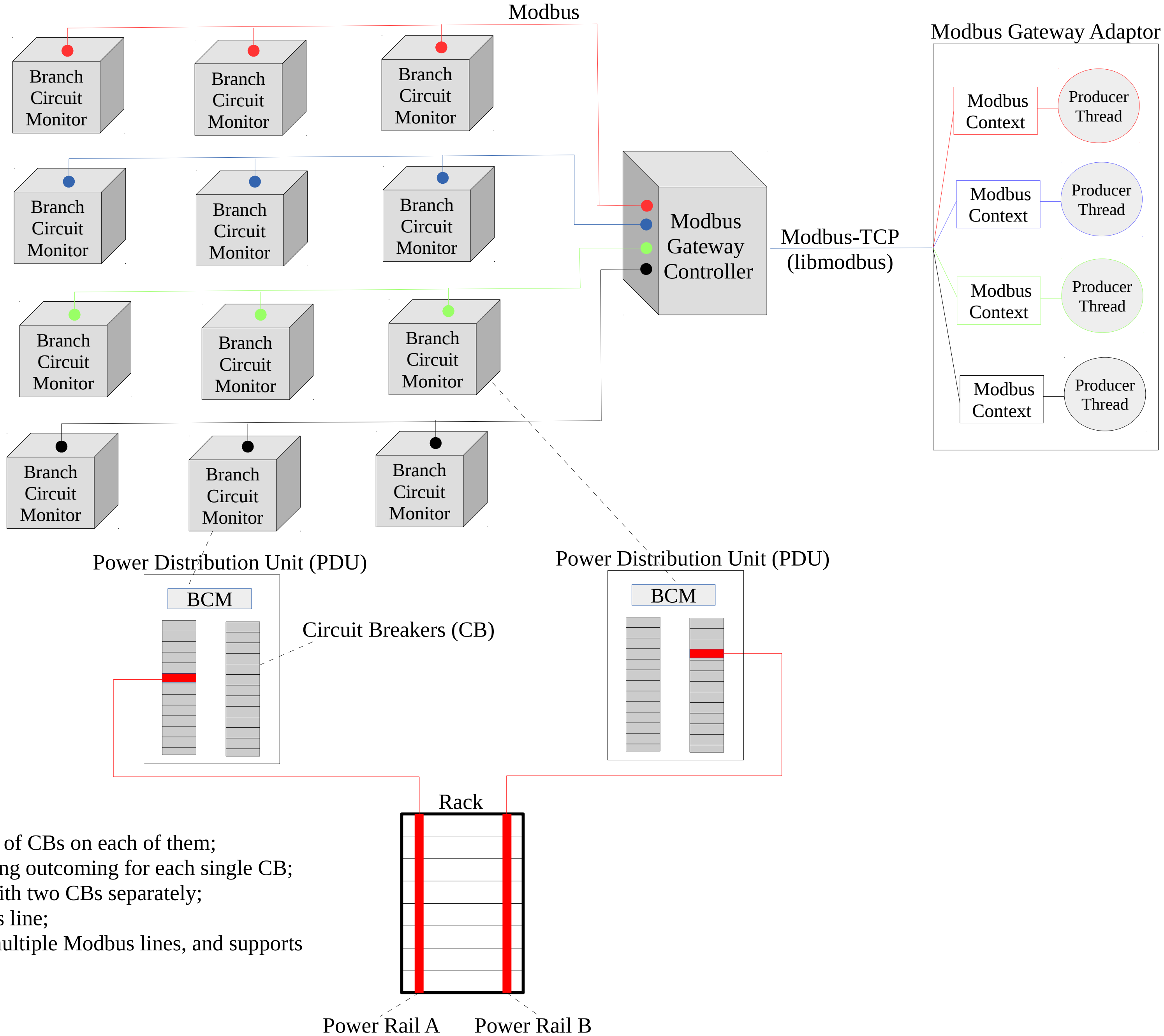
In the third place, if the to-be-deleted Listener is the last one in the queue, the client application will also request oBIX server to delete relevant watch object and then destroy relevant worker thread and Device specific CURL handles on the client side. In such case no worker thread should ever be running at back-end when relevant API returns. To this end, the deletion thread in scenario #2 needs to wait for the completion of the running worker thread before terminating it, which in turn requires the Device's mutex protecting the Listeners queue must be released by the deletion thread in order to avoid a deadlock with the worker thread who also competes on the same mutex.

However, above method invites a race condition. Once the Device's mutex is released by the deletion thread, another application thread in scenario #1 may be scheduled in immediately which tries to add a new Listener into the queue (and a new watchItem to relevant watch object on the server side). However, the entire watch object and relevant structures will be deleted shortly after when the mutex is released and the deletion thread is resumed.

To further address this race condition, a flag indicating whether the watch object on the server side is under deletion and a conditional are adopted. Any thread in scenario #1 requesting to create a new watch object would have to wait for the completion of the deletion of an existing one.

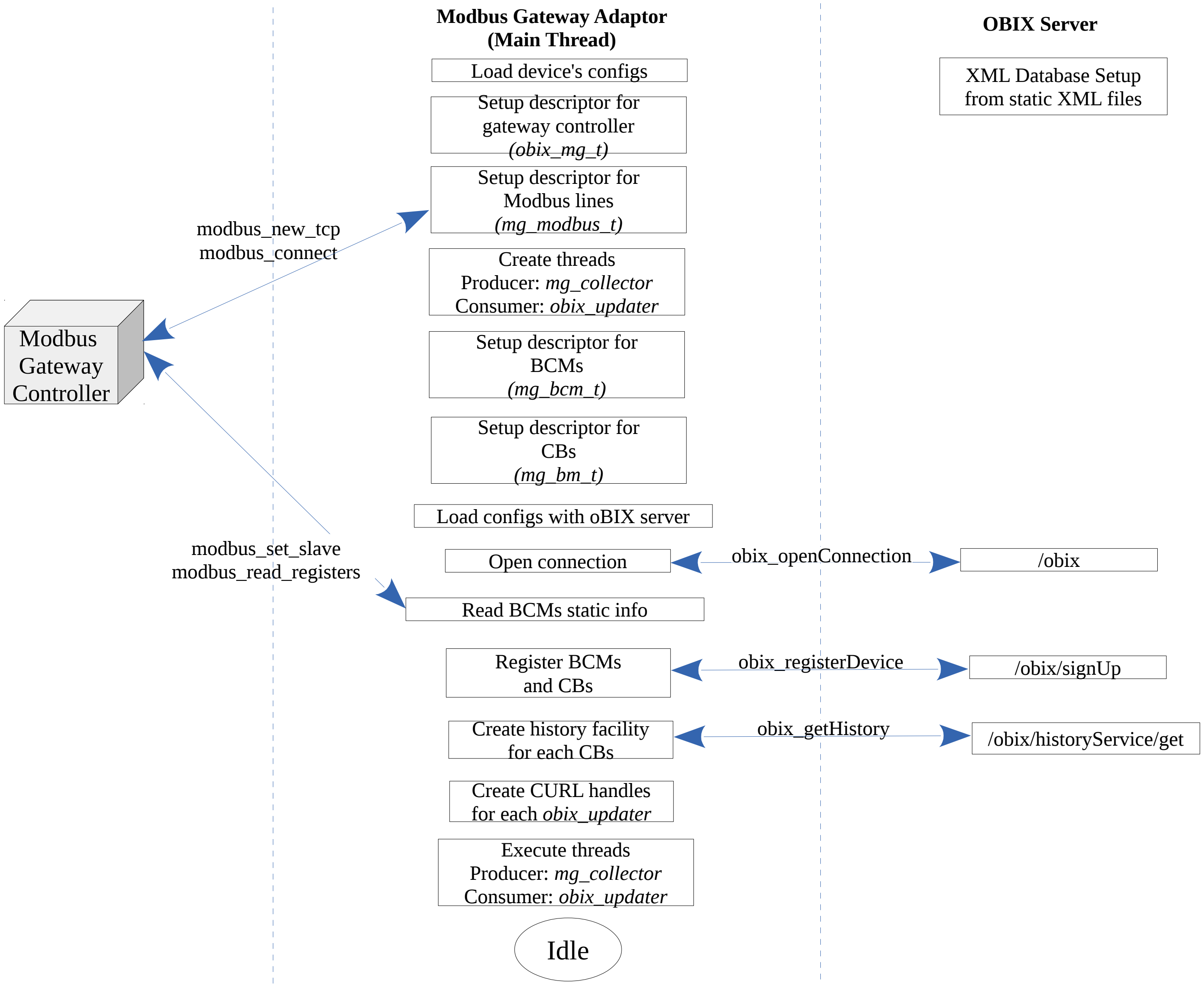
6. Modbus Gateway Adaptor

6.1 Hardware Connection



1. One PDU hosts two panels, with dozens of CBs on each of them;
2. One BCM embeds inside a PDU, metering outcoming for each single CB;
3. The two power rails of a rack connect with two CBs separately;
4. BCMs are daisy-chained on one Modbus line;
5. One Modbus gateway controller hosts multiple Modbus lines, and supports concurrent access to each of them.

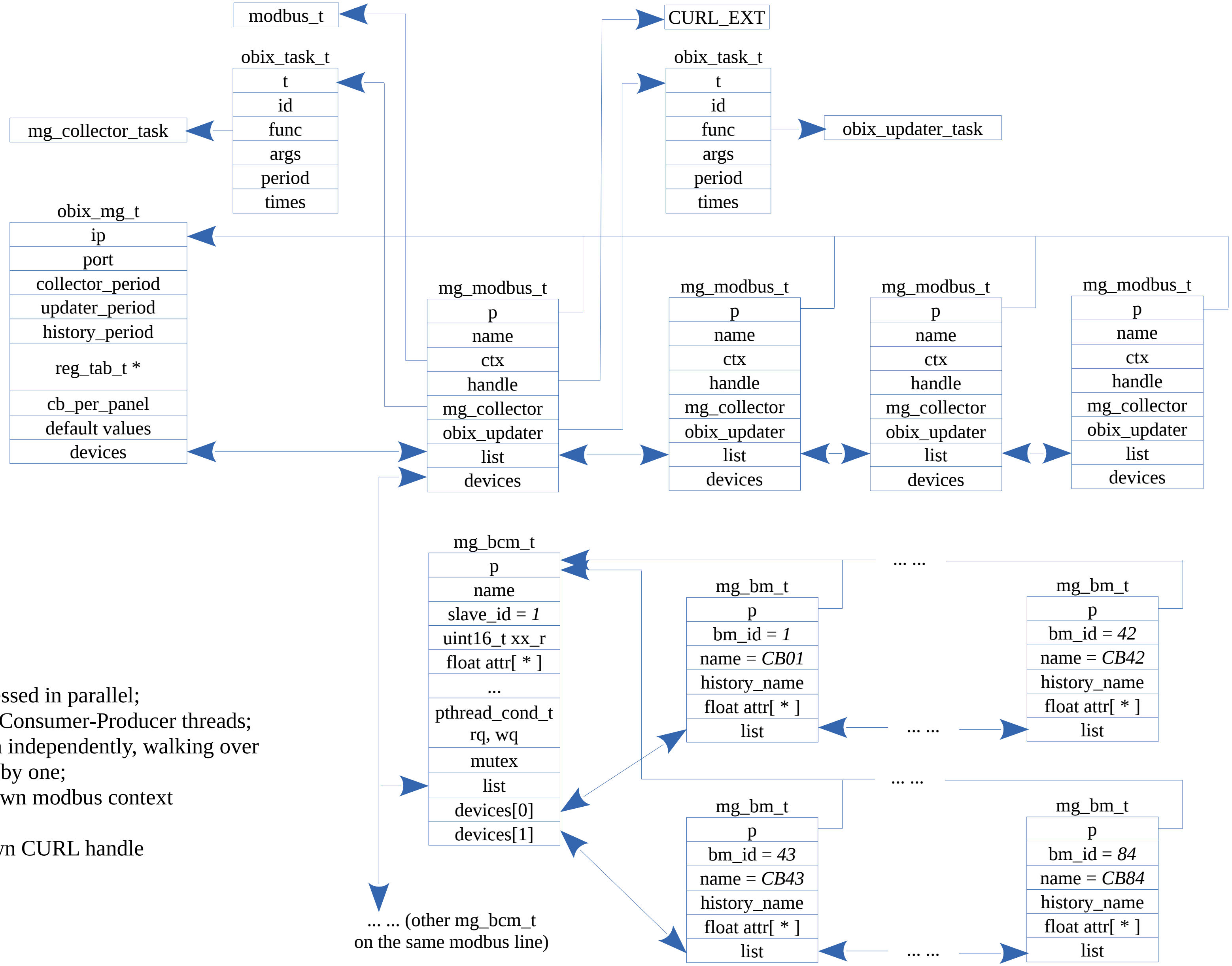
6.2 Interaction between Hardware and Software



6.3 oBIX Contracts for a BCM and a BM

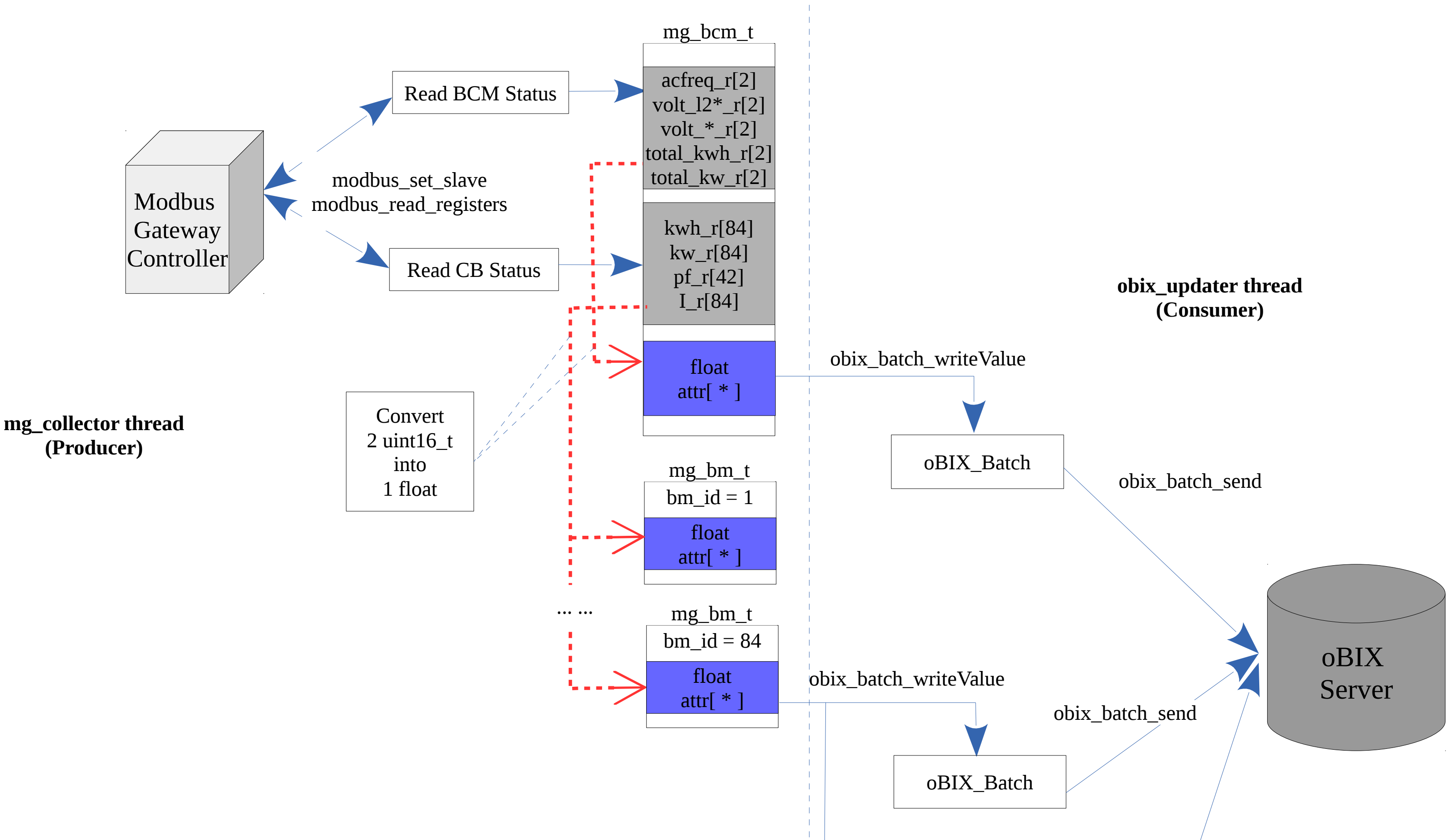
```xml
<obj name="4A-1A" href="/obix/deviceRoot/M1/DH4/4A-1A" is="nextdc:VerisBCM">
   <int name="SlaveID" href="SlaveID" val="1"/>
   <int name="SerialNumber" href="SerialNumber" val="0x4e342ef9" writable="true"/>
   <int name="Firmware" href="Firmware" val="0x03ed03f4" writable="true"/>
   <int name="Model" href="Model" val="15172" writable="true"/>
   <int name="CTConfig" href="CTConfig" val="2" writable="true"/>
   <str name="Location" href="Location" val="AUDM1DH4 PDU-4A-1A Panel #1" writable="true"/>
   <real name="ACFreq" href="ACFreq" val="50.000000" writable="true"/>
   <real name="VoltL-N" href="VoltL-N" val="242.080078" writable="true"/>
   <real name="VoltL-L" href="VoltL-L" val="418.952057" writable="true"/>
   <real name="VoltA" href="VoltA" val="240.157227" writable="true"/>
   <real name="VoltB" href="VoltB" val="243.659668" writable="true"/>
   <real name="VoltC" href="VoltC" val="242.563507" writable="true"/>
   <real name="kWh" href="kWh" val="218.000000" writable="true"/>
   <real name="kW" href="kW" val="0.000000" writable="true"/>
   <real name="CurrentAverage" href="CurrentAverage" val="0.000000" writable="true"/>
   <abstime name="LastUpdated" href="LastUpdated" val="2014-05-19T01:19:24Z" writable="true"/>
   <bool name="Online" href="OnLine" val="true" writable="true"/>
   <list name="Meters" href="Meters" of="nextdc:Meter">
      <obj name="CB01" href="CB01" is="nextdc:Meter">
         <real name="kWh" href="kWh" val="25.444157" writable="true"/>
         <real name="kW" href="kW" val="0.000000" writable="true"/>
         <real name="V" href="V" val="240.157227" writable="true"/>
         <real name="PF" href="PF" val="0.900000" writable="true"/>
         <real name="I" href="I" val="0.000000" writable="true"/>
      </obj>
       ......
      <obj name="CB84" href="CB84" is="nextdc:Meter">
         <real name="kWh" href="kWh" val="50.943935" writable="true"/>
         <real name="kW" href="kW" val="0.000000" writable="true"/>
         <real name="V" href="V" val="243.659668" writable="true"/>
         <real name="PF" href="PF" val="0.900000" writable="true"/>
         <real name="I" href="I" val="0.000000" writable="true"/>
      </obj>
   </list>
</obj>
```

6.4 Software Infrastructure



1. Different modbus lines are accessed in parallel;
2. Each modbus line has a pair of Consumer-Producer threads;
3. Consumer-Producer threads run independently, walking over BCMs on one modbus line one by one;
4. Each Consumer thread has its own modbus context (which is not thread safe);
5. Each Producer thread has its own CURL handle

6.5 Producer-Consumer Model



1. For each BCM, the Producer thread first reads its registers via the modbus gateway controller;
2. The Producer thread then converts the data into 32bit floats, and distributes for each CB;
3. The Consumer thread uses a batch object to carry the latest data of a BCM or CB, then send it to the oBIX server to have relevant XML object there updated;
4. At specified pace, the Consumer thread appends a history record for each CB.