Tech deep-dives of the oBIX Server and Adaptor

oBIX 1.0 Release

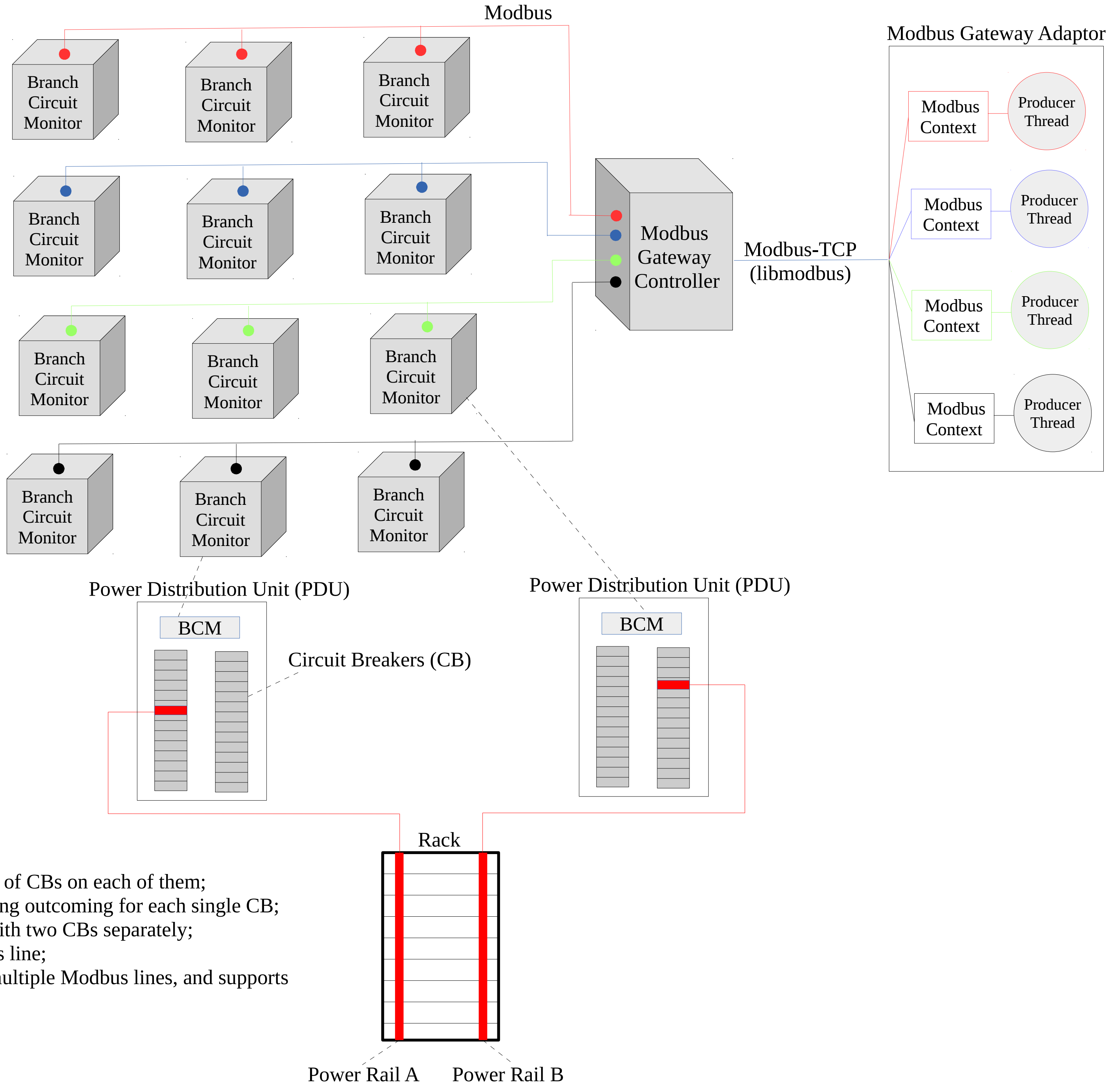
Qingtao Cao [harry.cao@nextdc.com]

Last updated: 2014-7-3

Table of Content

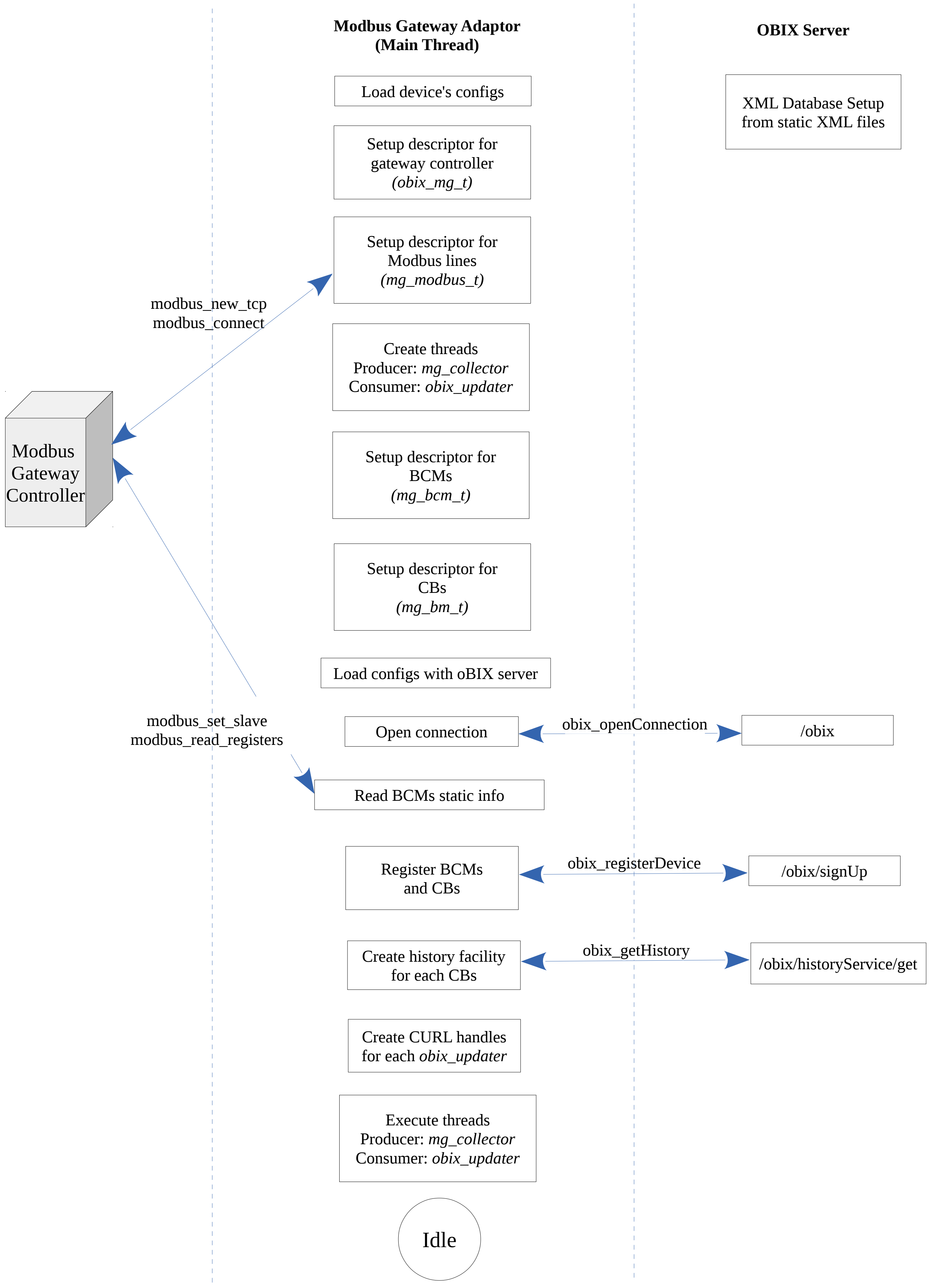
Source code: git clone git@github.com:ONEDC/obix.git

Hardware Environment of the oBIX Modbus Gateway Adaptor



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.

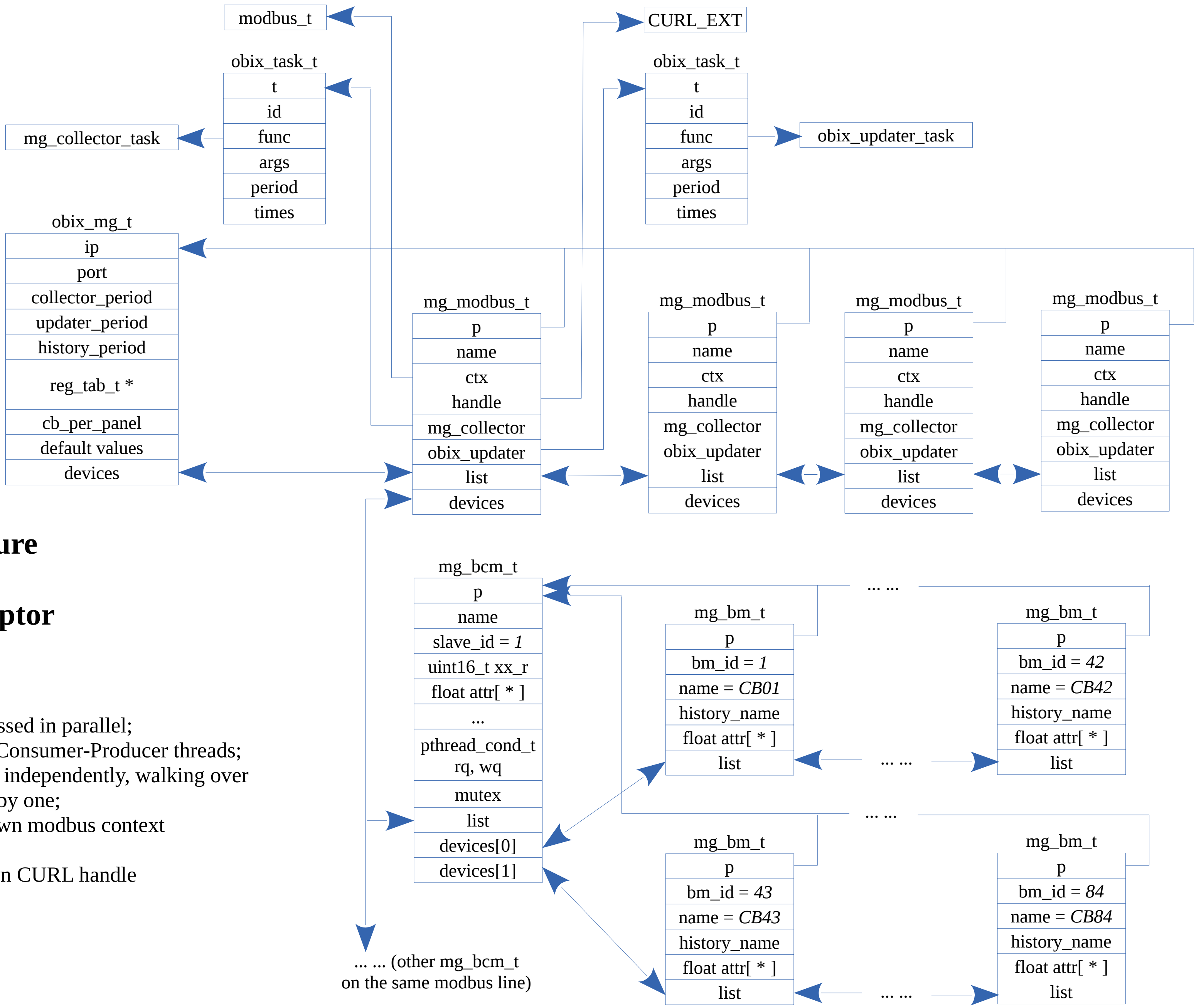
Interaction between
HW and SW of the oBIX Modbus Gateway Adaptor



Modbus Gateway Adaptor (Main Thread)
OBIX Server
Load device's configs
XML Database Setup from static XML files
Setup descriptor for gateway controller (obix_mg_t)
Setup descriptor for Modbus lines (mg_modbus_t)
modbus_new_tcp
modbus_connect
Create threads Producer: mg_collector Consumer: obix_updater
Modbus Gateway Controller
Setup descriptor for BCMs (mg_bcm_t)
Setup descriptor for CBs (mg_bm_t)
Load configs with oBIX server
modbus_set_slave
modbus_read_registers
Open connection
obix_openConnection
/obix
Read BCMs static info
Register BCMs and CBs
obix_registerDevice
/obix/signUp
Create history facility for each CBs
obix_getHistory
/obix/historyService/get
Create CURL handles for each obix_updater
Execute threads Producer: mg_collector Consumer: obix_updater
Idle

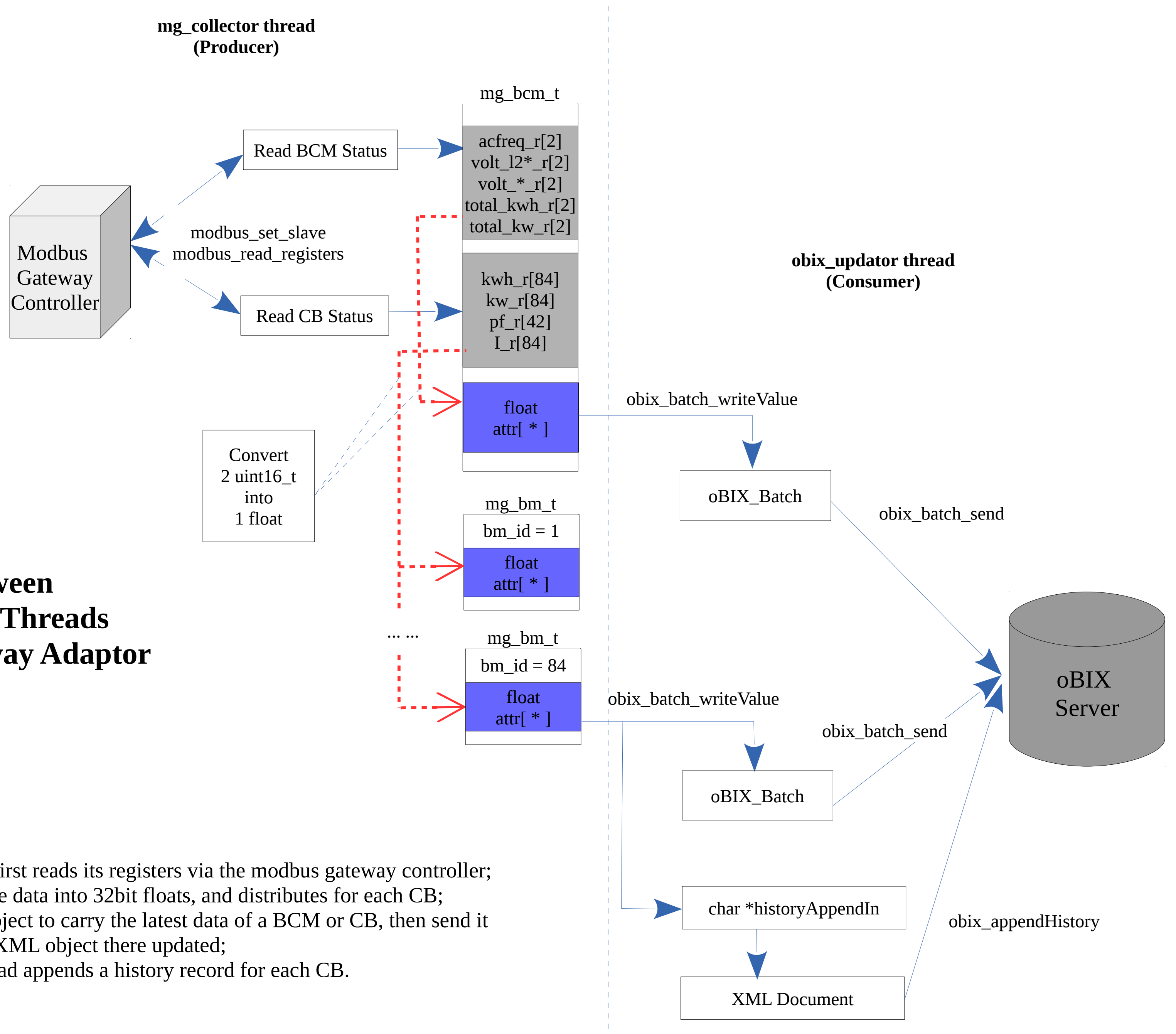
XML Contracts for the BCM and CB used in the oBIX Modbus Gateway Adaptor

```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="2014-05-19T01:19:24" 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="CB02" 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>
```



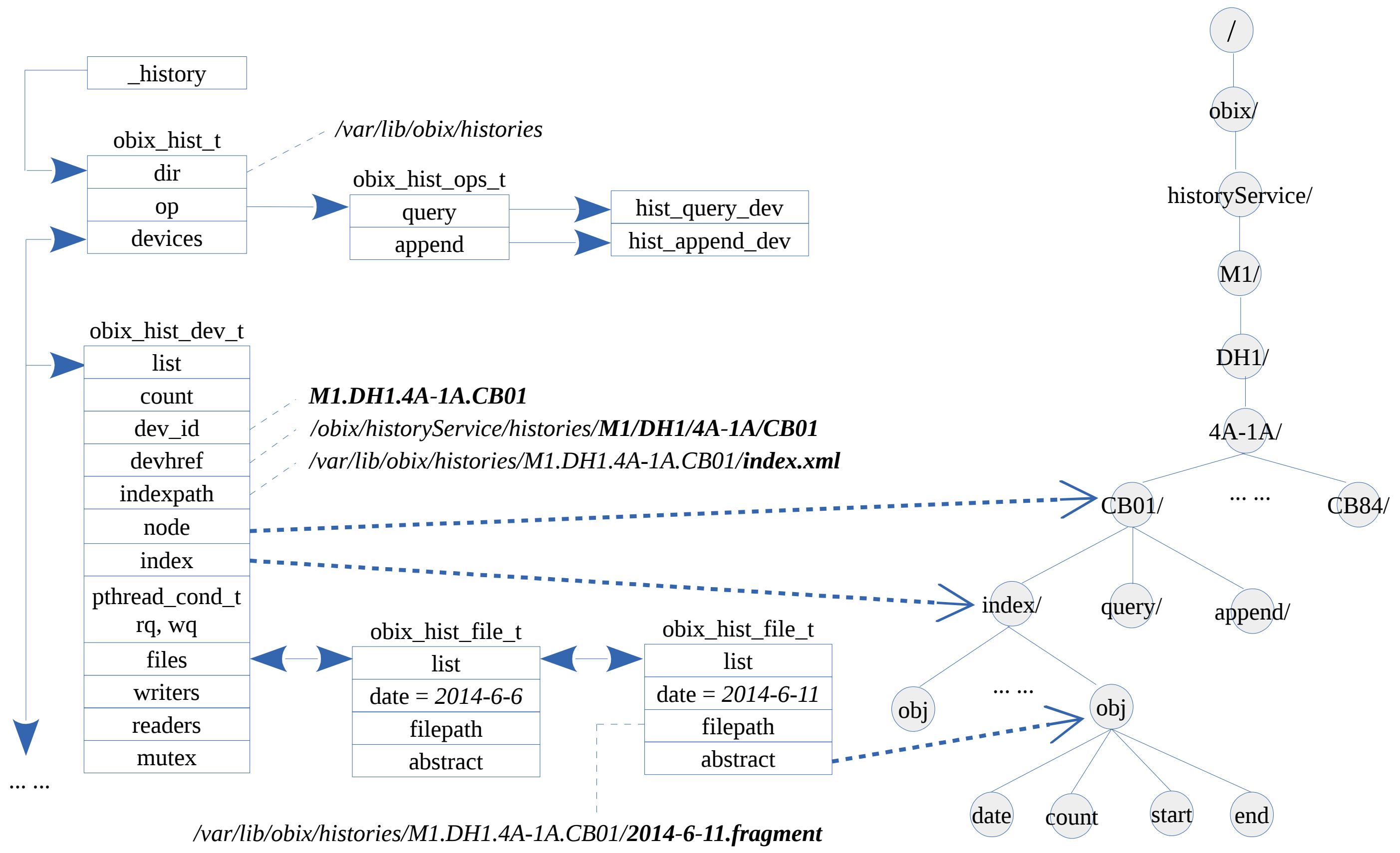
**Software Infrastructure
of the oBIX
Modbus Gateway Adaptor**

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 (which is not thread safe);



The Interaction between the Producer-Consumer Threads Of the oBIX Modbus Gateway Adaptor

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.



```
/var/lib/obix/histories/
    ├─── M1.DH1.4A-1A.CB01
 ... ...   ├──── 2014-06-06.fragment
           ├──── 2014-06-11.fragment
           └──── index.xml
```

```
<obj/>
......
<obj is="obix:HistoryRecord">
  <abstime name="timestamp" val="2014-06-11T07:40:05"/>
  <real name="kWh" val="0.000000"/>
  <real name="kW" val="0.000000"/>
  <real name="V" val="230.000000"/>
  <real name="PF" val="0.900000"/>
  <real name="I" val="0.000000"/>
</obj>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<list name="index" href="index" of="obix:HistoryFileAbstract">
  <obj is="obix:HistoryFileAbstract">
    <date name="date" val="2014-06-06"/>
    <int name="count" val="577"/>
    <abstime name="start" val="2014-06-06T01:51:33"/>
    <abstime name="end" val="2014-06-06T06:39:02"/>
  </obj>
  <obj is="obix:HistoryFileAbstract">
    <date name="date" val="2014-06-11"/>
    <int name="count" val="810"/>
    <abstime name="start" val="2014-06-11T00:35:49"/>
    <abstime name="end" val="2014-06-11T07:40:05"/>
  </obj>
</list>
```

Software Infrastructure
of the History Subsystem in the oBIX Server

2.2 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 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 nothing but 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 moment 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 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. Therefore oBIX clients can't append a history record with timestamp older than that of the very end 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.

2.3 XML DOM Hierarchy

The middle part of the 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 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.

2.4 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 imposes no limitation on the number of history facilities on the oBIX server.

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.

2.5 Multi-thread Support

In the first place, different history facilities are allowed to be accessed in parallel.

Moreover, in order to support concurrent access on one history facility from multiple oBIX clients who may query from and append to it simultaneously, the obix_hist_dev_t descriptor records its state information and employs POSIX pthread mutex and conditionals to synchronise among readers and writers.

In current implementation, the number of existing readers and the number of waiting writers are counted and multiple readers are permitted to read from one history facility in parallel. However, writers are excluded from each other and from any readers. Furthermore, no writers will suffer from starvation since no more readers will be allowed if there is any writers waiting for the completion of existing readers, in which case new readers will have to wait until each waiting writer has a chance to finish its task first.

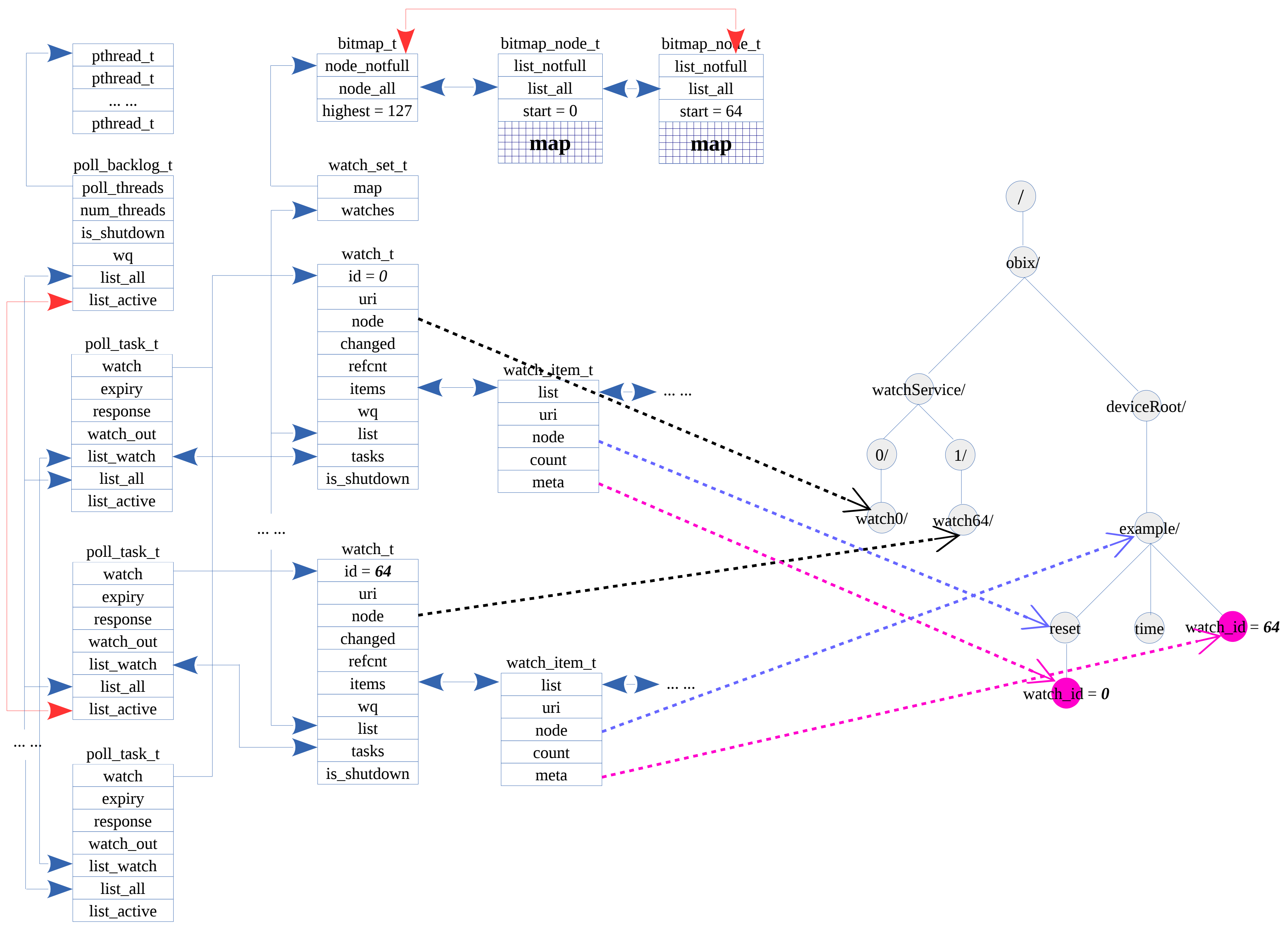
2.6 Performance of the Query Operation

Having said earlier, a history facility may contain unlimited 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 needs to specify a limit on the number of records 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.

Software Infrastructure of the Watch Subsystem in the oBIX Server



3.2 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 href contains all XML contracts for every single watch objects already created on the oBIX server. Each watch object has a unique ID number and no more than 64 watch objects are grouped under one parent href. 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 href, it would take ages to traverse the entire watchService sub-tree 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 all XML contracts registered by different 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 which watch object should be notified when changes occur.

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 the rest of the example device other than the reset node is 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.

3.3 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 any watch object is deleted, its ID number is recycled and reused properly.

Whenever a brand-new watch object is created upon a request from an oBIX client, 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. This way 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 href, while the “node” pointer in the watch_item_t descriptor points to the monitored XML node. These pointers are represented by bold, dotted, black or blue lines with arrow in the above diagram.

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. In the above diagram they are represented by bold, dotted and pink lines with arrow.

3.4 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 blocking 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 FCGI response structure which will carry 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 will 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.

Moreover, 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.

The watch subsystem employs POSIX pthread mutex to eliminate potential race conditions among polling threads, utilises pthread conditionals to synchronise among polling threads and the oBIX server thread, and also manipulates a reference count to ensure a watch_t descriptor won't get destroyed until all relevant pending poll tasks have been properly handled.

3.5 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, resulting in 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 also 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.